Federating IoT and cloud infrastructures to provide scalable and interoperable Smart Cities applications, by introducing novel IoT virtualization technologies

EU Funding: H2020 Research and Innovation Action GA 814918; JP Funding: Ministry of Internal Affairs and Communications (MIC)

Deliverable 3.2
Cloud Oriented Services - Second Release

Deliverable Type:	Report
Deliverable Number:	D3.2
Contractual Date of Delivery to the EU:	28.2.2021
Actual Date of Delivery to the EU:	28.2.2021
Title of Deliverable:	Public
Work package contributing to the Deliverable:	WP3
Dissemination Level:	Public
Editor:	Hajime Tazaki, Martin Bauer
Author(s):	Hajime Tazaki (IIJ); Martin Bauer and Bin Cheng (NEC); Andrea Detti and Giuseppe Tropea (CNIT); Juan Andrés Sanchez, Juan Antonio Martinez and Antonio Skarmeta (OdinS); Hidenori Nakazato, Kenji Kanai and Hidehiro Kanemitsu (Waseda); Hiroaki Mukai (KIT)
Internal Reviewer(s):	Andrea Detti (CNIT)
Abstract:	This deliverable reports the second version of the Fed4IoT cloud-oriented services
Keyword List:	IoT virtualization, Light Compute Virtualization, ICN, FogFlow

Disclaimer

This document has been produced in the context of the EU-JP Fed4IoT project which is jointly funded by the European Commission (grant agreement n° 814918) and Ministry of Internal Affairs and Communications (MIC) from Japan. The document reflects only the author's view, European Commission and MIC are not responsible for any use that may be made of the information it contains

Table of Contents

List of Figures

List of Tables

Fed4IoT Glossary

Table 1 lists and describes terms relevant to this deliverable.

Term	Definition
FogFlow	An IoT edge computing framework that automatically orchestrates dynamic data processing flows over cloud- and edge- based infrastructures. Used for ThingVisor development.
Flexible Compute Virtualization	The Fed4IoT compute virtualization platform able to manage heterogeneous virtualization technologies (Docker, unikernel, etc.) over cloud- and edge- based infrastructures. Used for deployment of VirIoT components.
Information Centric Networking	New networking technology based on named contents rather than IP addresses. Used for ThingVisor development.
IoT Broker	Software entity responsible for the distribution of IoT information. For instance, Mobius, Orion and Scorpio, can be considered as Brokers of oneM2M, NGSI and NGSI-LD IoT platforms, respectively.
Neutral Format	IoT data representation format that can be easily translated to/from the different formats used by IoT Brokers.
Real IoT System	IoT system formed by real (as opposite to virtual) things whose data is exposed trough a Broker.
System DataBase	Database for storing system information.
ThingVisor	System component that implements Virtual Things.
VirIoT	Fed4IoT platform providing virtual IoT systems as a service.
Virtual Silo	Isolated virtual IoT system formed by Virtual Things, Brokers and Controllers.
Silo Controllers	System component working within a Virtual Silo.
Virtual Silo Flavour	Image of a specific Virtual Silo, e.g. a "Mobius flavour" is a Virtual Silo with a Mobius broker, a "MQTT flavour" is a Virtual Silo with a MQTT broker, etc.
Virtual Thing	An emulation of a real thing obtained by processing and/or controlling data of real things.
Tenant	User that accesses the Fed4IoT VirIoT platform to develop IoT applications.

Table 1: Fed4IoT Dictionary

1 Introduction

1.1 Purpose of the Document

This deliverable describes the Fed4IoT cloud-oriented services, which are positioned, within the whole Fed4IoT architectural framework, as shown in Figure 1.

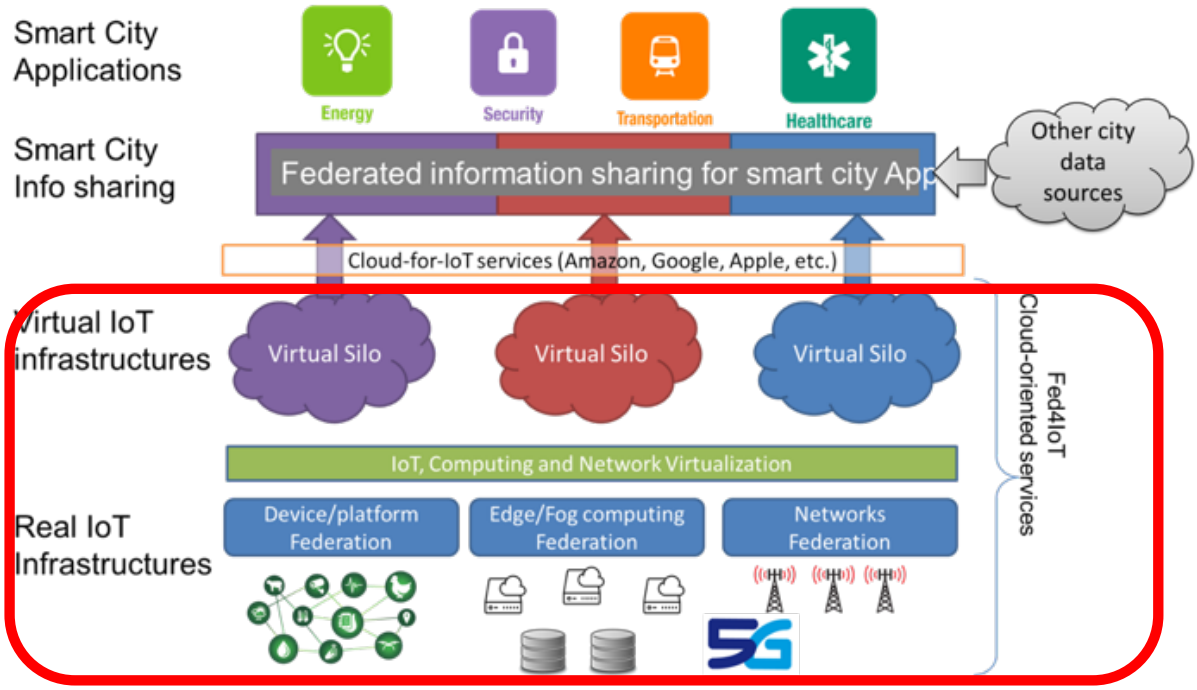


Figure 1: Fed4IoT Architectural Framework

1.2 Executive Summary

This deliverable presents the second release of Fed4IoT cloud-oriented services. First, we present the whole Fed4IoT virtualization stack (Section 2) composed of i) an upper platform, named VirIoT, proving IoT Virtualization services and ii) a lower computing virtualization platform, we named Flexible Compute Virtualization. Then, we provide details about VirIoT, in terms of REST API, CLI, control procedures, developed components (Section 3). We present optimization and design activities concerning specific VirIoT core components, i.e. ThingVisors (Section 4). Finally, we describe the underlying compute virtualization architecture we used, which is based on Kubernetes with virtualization solutions alternative to Docker, to address either running low-power devices or specific application needs (Section 5).

1.3 Quality Review

The internal Reviewer responsible for this deliverable is Andrea Detti (CNIT).

Version Control Table			
V.	**Purpose/Changes**	**Authors**	**Date**
0.1	ToC Revision from D3.1	Hajime Tazaki (IIJ)	21/1/2021
0.2	First contribution round	ALL	9/2/2021
0.3	Last contribution round	ALL	21/2/2021
0.4	Final review	Andrea Detti (CNIT)	28/2/2021

Table 2: Version Control Table

1.4 Major Updates from Previous Version D3.1

- Updated Section 2 to include latest evolution of the whole stack.
- Updated Section 3 to include latest evolution of the VirIoT platform (e.g. HTTP distribution system, Kubernetes support) and revised presentation of API, CLI and Control Commands.
- Changes in FIWARE ThingVisors (Section 3.3.2, 3.3.4) parameters.
- Removed Smart Parking ThingVisor (old section 3.3.1.3 in D3.1). Smart Parking use case requires two new ThingVisors: Parking Site ThingVisor (Section 3.3.4) and Regulated Parking Zones ThingVisor (Section 3.3.5)
- New Actuator ThingVisor which supports sending commands to a remote Context Broker (Section 3.3.7).
- New Wildlife Monitoring ThingVisor (Section 3.3.8).
- New CameraBot ThingVisor (Section 3.3.12) and FaceRecognition ThingVisor (Section 3.3.11).
- Orion NGSIv2 Flavour (Section 3.4.2) now supports sending commands upstream VirIoT components.
- New generalized NGSI-LD Flavour, able to accommodate any standard NGSI-LD-compliant IoT Broker (Section 3.4.4).
- Updated Section 4.1 on FogFlow-based ThingVisor Factory with added use cases (Section 4.1.1), decentralized orchestration mechanism (Section 4.1.5), and some evaluation results (Section 4.1.7).
- Updated Section 4.2, which was previously titled “Service Function Chaining” and is now “VirIoT ThingVisor Factory.” The section for service function chaining is renamed to “ThingVisor Chaining” and is moved to Section 4.2.2. Other sections under Section 4.2 are new in this revision.
- Extension of the Flexible Compute Virtualization architecture towards distributed deployments based on Kubernetes (Section 5.1).

- Added extended description of resource minimization and Kubernetes integration on Linux Kernel Library (LKL, in Section 5.4).

2 The Fed4IoT Virtualization Stack

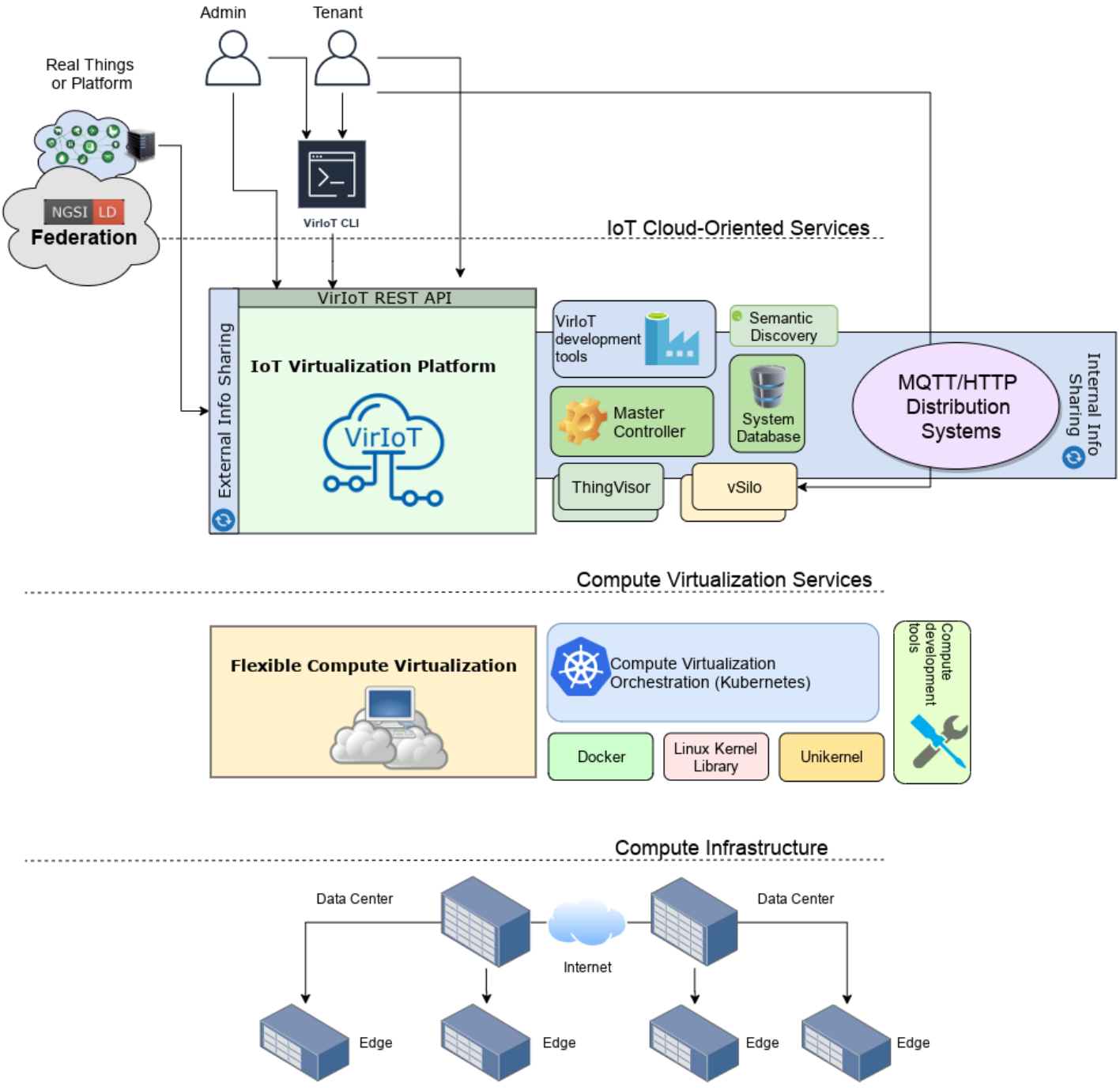


Figure 2: Fed4IoT Virtualization Stack

Figure 2 shows the stack of Fed4IoT's IoT virtualization services. Starting from the top of the figure, we have the Fed4IoT IoT Cloud-Oriented Services, which offer IoT Infrastructure-as-a-Service: an IoT infrastructure, namely a Virtual Silo, which is an isolated environment, exchanges data items with a configurable set of Virtual Things,

and this information is exposed through a IoT Broker of choice, inside the Virtual Silo. These IoT Cloud-Oriented Services are implemented by the VirIoT platform, which we introduced in D2.3, and we now detail in Section 3 in terms of REST APIs, CLI, control procedures, components [1]. VirIoT exposes both a REST API and a Command Line Interface (CLI) to Administrators and Tenants, and also an External Info Sharing interface to other IoT platforms that wish to exchange their information with VirIoT (e.g. provide ThingVisors with their data) and/or create a NGSI-LD federation which includes information coming from VirIoT.

Additionally, we design VirIoT-tailored development tools, named ThingVisor Factories, that simplify the development and deployment of complex and distributed ThingVisors. These additional VirIoT development tools are based on FogFlow and on the concept of ThingVisor chaining, and they are described in Section 4.

The VirIoT platform exploits compute virtualization services to deploy and run its components that we based on Kubernetes. In turn, compute virtualization services are based on a distributed cloud/edge computing infrastructure. The compute virtualization architecture is flexible, i.e. can use different virtualization technologies (e.g., different container formats) to best fit the needs and constraints of the service and of the host device. For instance, for devices such as servers, Docker is the reference virtualization technology. For low-power devices (e.g. Raspberry PI) deployed at the network edge, the Kubernetes nodes can use other innovative technologies the project is exploring such as Linux Kernel Libraries and Unikernels, which can be built by using Unikraft tools. Compute Virtualization Technologies are described in Section 5.

[1] Information models and networking details are in D4.2

3 IoT Cloud-Oriented Services

Fed4IoT's cloud-oriented services are implemented by a platform named VirIoT, which can be defined as a *Cloud of Things*. As shown in Figure 3, VirIoT is an open-source micro-services architecture composed by many components, also named *services*, such as ThingVisors, vSilos, vThings, etc. VirIoT provides tenants (users) with Virtual Silos (vSilos), which are virtual isolated IoT Infrastructures connected to Virtual Things (vThings) [2].

A Virtual Thing is an abstraction implemented by a ThingVisor, which exploits external information coming from real IoT infrastructures to create the illusion a thing is present [2], by producing the data and provoking the actions it would have produced or provoked, were it real. Virtual Things handle two kinds of data: *generic contents* and *context data*:

- Generic contents are heterogeneous (large) data pieces, such as images or video streams exposed as HTTP resources. A tenant accesses these contents, adding the vThing to her vSilo and through the mediation of an *HTTP Broker* included in her vSilo.

- Context data pieces are a few bytes long and digitally represent a (real or virtual) thing with its properties, such as the status of a lamp or the temperature of a room. A tenant accesses these contents by adding the vThing to her vSilo and through the mediation of an *IoT Broker* included in her vSilo. The technology of the IoT Broker is of user's choice (e.g. the oneM2M Mobius server, the NGSIv2 Orion broker developed by FIWARE, or the new Scorpio NGSI-LD Context Broker by NEC, etc.)

Virtual Things can be Virtual Sensors or Virtual Actuators (see D4.2 for their respective information models). A Virtual Sensor is a vThing that only has sensing capabilities and produces context data and generic HTTP contents. A Virtual Actuator is a vThing that has actuation capabilities too, therefore not only produces data but also consumes data sent by users to control its status, perform actions, etc[3].

For instance we can have an external infrastructure of real cameras connected to the VirIoT system, and a virtual "person counter" (virtual sensor), counting the number of people in a room in real-time, thus producing "virtual" measurements (context data) that are generated by processing data coming from the infrastructure. As another example, a virtual "face recognition" is a virtual actuator that a tenant can configure in order to be notified when a specific face is detected in the current frame and recognized as one of the designated target pictures. This vThing can use the same camera used by the virtual person counter and it is considered an actuator since its "behaviour" (which face to recognize) is controlled by the user.

A tenant can create a Virtual Silo and add to it (i.e., connect to it) its preferred set of Virtual Things, among the ones offered by VirIoT. Virtual Silos differ each other both by the set of connected Virtual Things and by the IoT Broker technology they use to expose

[2] similar to the illusion created by an Hypervisor when running a virtual machine

[3] Many actuators also have properties that can be monitored, such as the current on/off status of a lamp. For this reason, we consider that actuators may also include sensing functions.

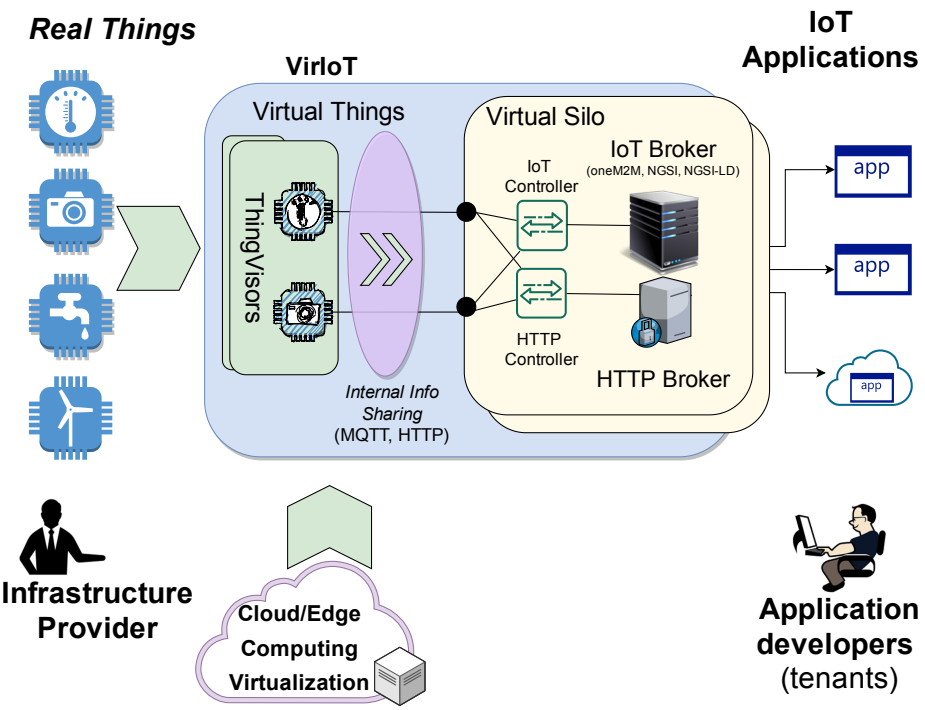


Figure 3: VirIoT services

vThing context data. When a tenant creates a Virtual Silo, the VirIoT platform runs a new instance (Linux container) of a so called Virtual Silo Flavour (container image), which is an empty image of the Virtual Silo, only containing the IoT/HTTP Brokers and related Controllers. Such controllers are local agents interacting with the rest of the VirIoT architecture components (Master Controller, ThingVisors, etc.). The Controllers configure their respective Brokers, and the IoT controller also carries out information model translation for context data, as described in D4.2. As soon as a Virtual Silo is up and running, the tenant can connect to it the preferred Virtual Things.

As reported in D2.3, the VirIoT architecture is formed by the following components:

1. Master Controller: the main orchestration component, exposing a REST API to the tenants and to the administrator, and in turn interacting with other components, to manage ThingVisors and Virtual Silos. A python CLI, too, can be used to interact with the Master controller.

2. ThingVisors: a pluggable set of components, each of them implementing one or more Virtual Things.

3. Internal information sharing system: a data distribution service used to transfer, within the platform boundaries, the control messages and the data generated by the Virtual Things. It is actually made by two Data Distribution Systems based on MQTT and HTTP, respectively. The MQTT one implements a distributed MQTT system used to transfer vThing context data and VirIoT control messages. The HTTP distribution system resembles a content delivery network made of HTTP proxies with caching capabilities, that is used to transfer (usually large size) vThing generic contents.

4. External information sharing system: custom interfaces used to exchange data between ThingVisors and external IoT platforms.

5. Image repositories: image stores (e.g. DockerHub) hosting base images of Virtual Silos (Flavours) and ThingVisors.

6. Virtual Silos: instances of a Virtual Silo Flavour that include HTTP and IoT Controllers, and IoT and HTTP Brokers, and that interacts with a set of Virtual Things, as selected by the tenant.

7. System DB: database containing system information, mainly used to keep system components as much as possible stateless.

8. ThingVisor Factories: tools for developing and orchestrating the internal components of complex and distributed ThingVisors.

9. Compute Virtualization Layer: a compute virtualization architecture, exposing services for deploying different types of images (Docker, Unikernel, etc.) of the VirIoT components on a distributed and heterogeneous cloud platform, formed by central and edge resources.

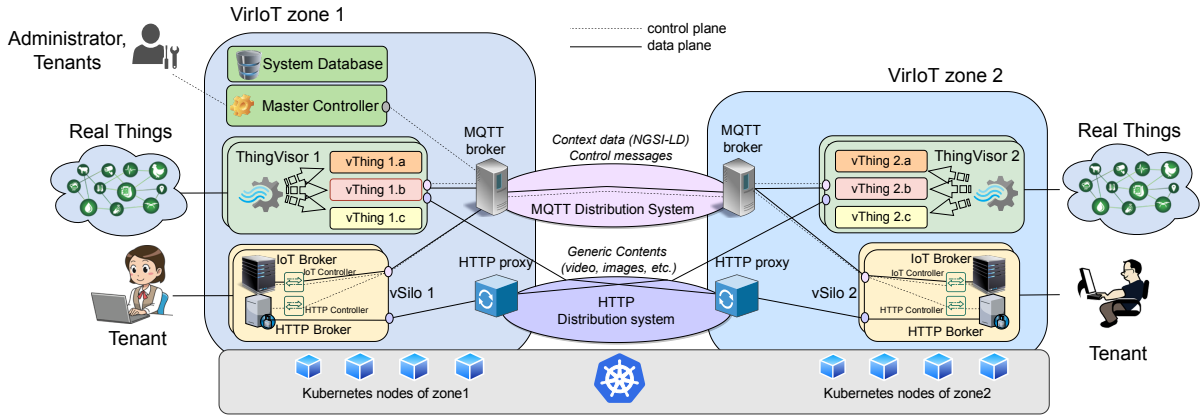


Figure 4: VirIoT System Architecture

As shown in Figure 4, the latest implementation of the architecture uses a Kubernetes (K8S) cluster[4] as its underlying Compute Virtualization Layer, possibly spreading over a wide geographical area of central and edge data centers. The K8S cluster is organized in *zones*. A zone can be a data center, or an edge node. The nodes have specific K8S labels (`viriot-zone`) that allow to discriminate them and to control where to install each VirIoT service. The container environments have been extended to support the light Compute Virtualization Technology (Unikernels, Linux Kernel Library) the project is proposing to enable deployment of the VirIoT components on edge/fog low-power devices (see Section 5). The platform offers edge-computing functionality, meaning that it is possible to control *where* ThingVisors and vSilos can be deployed. This way, tenants can deploy their vSilos at a specific data center closer to the final applications, with

[4]Kubernetes is a widespread container orchestration platform. Its deployment unit is named POD, which is a group of Linux containers [1].

obvious advantage in terms of latency and bandwidth consumption. For the same reason, the administrator can deploy ThingVisors near the real things that interact with them.

In what follows we describe the VirIoT API/CLI used by tenants and administrator to control the platform. Then we describe the internal procedures that must be followed by developers of new vSilos and ThingVisors to integrate them in the platform. The information models used to exchange data between ThingVisors and vSilos are detailed in D4.2. In this deliverable we only describes the content of control messages used by internal procedures.

3.1 VirIoT REST API and Command Line Interface

The following several sections give full details of the API of the VirIoT platform, available both through a RESTful option and through a command line option. Most of the time-consuming API calls follow a "fire and forget" approach, where commands are started by invoking the corresponding API endpoint, and they are asynchronously executed in background by the platform. Thus, each such call is non-blocking and immediately returns with a status message telling the current status of the command execution. Progress of the command execution can be subsequently monitored through specific inspect API calls. Other API calls, such as user registration and login, complete synchronously and the response message immediately gives back the command result.

3.1.1 Registration

It registers a new user to the system. The administrator sends a request with a triplet, formed by the new user ID, the password and the role of the new user to be registered. The role can be either admin or regular user.

Needed privileges: Administrator.

3.1.1.1 CLI

```
f4i.py register [-h] [-c CONTROLLERURL] [-u USERID] [-p PASSWORD] [-r
    ROLE]
```

CLI Example

```
python3 f4i.py register -c http://127.0.0.1:8090 -u tenant1 -p password -
    r user
```

3.1.1.2 REST API

HEADERS:

Authorization:	Bearer ⟨*JSONWebToken*⟩
Accept:	application/json
Content-Type:	application/json
Cache-Control:	no-cache

POST /register

```
{
  "userID":[USERID],
  "password":[PASSWORD],
  "role":[ROLE]
}
```

RESPONSE:

```
{
  "message":[result of the registration: either success or fail]
}
```

3.1.2 Unregistration

It unregisters the user, deleting the credentials from the System DB.

Needed privileges: Administrator can unregister anybody. Regular users can only unregister themselves.

3.1.2.1 CLI

```
f4i.py unregister [-h] [-c CONTROLLERURL] [-u USERID]
```

CLI Example

```
python3 f4i.py unregister -c http://127.0.0.1:8090 -u tenant1
```

3.1.2.2 REST API

HEADERS:

Authorization:	Bearer ⟨*JSONWebToken*⟩
Accept:	application/json
Content-Type:	application/json
Cache-Control:	no-cache

POST /unregister

```
{
  "userID":[USERID]
}
```

RESPONSE:

```
{
  "message":[result of the unregistration: either success or fail]
}
```

3.1.3 Login

It lets a user login into the system. The Master Controller checks if the information given by the user is correct. In order to do so, it validates it against information fetched from the System DB. If the operation is successful, the user receives an `access token` that allows the user to perform subsequent CLI commands.

Needed privileges: User credentials are needed.

3.1.3.1 CLI

```
f4i.py login [-h] [-c CONTROLLERURL] [-u USERID] [-p PASSWORD]
```

CLI Example

```
python3 f4i.py login -c http://127.0.0.1:8090 -u tenant1 -p password
```

3.1.3.2 REST API

HEADERS:

Accept:	application/json
Content-Type:	application/json
Cache-Control:	no-cache

POST /login

```
{
  "userID":[USERID],
  "password":[PASSWORD]
}
```

RESPONSE in case of success (status 200):

```
{
  "access_token":[access token string],
  "role":[role of the user who logged in]
}
```

RESPONSE in case of failure (status 401):

```
{
  "message":"Bad username or password"
}
```

3.1.4 List Users

Used by administrator to get a list of all registered users. It gives back an array of JSON objects, each object giving information about a User.

Needed privileges: Administrator.

3.1.4.1 CLI

```
f4i.py list-users [-h] [-c CONTROLLERURL]
```

CLI Example

```
python3 f4i.py list-users -c http://127.0.0.1:8090
```

3.1.4.2 REST API

HEADERS:

Authorization: Bearer ⟨*JSONWebToken*⟩
Accept: application/json
Content-Type: application/json
Cache-Control: no-cache

GET /listUsers

```
[Body is empty, token is in Headers]
```

RESPONSE:

```
[
  {JSON object with information of a User}
]
```

3.1.5 Logout

It logs out a user from the system. The `access_token` of the specified user is added to a local black-list, that will be used to deny further operations using the aforementioned token.

Needed privileges: All users.

3.1.5.1 CLI

```
f4i.py logout [-h] [-c CONTROLLERURL]
```

CLI Example

```
python3 f4i.py logout -c http://127.0.0.1:8090
```

3.1.5.2 REST API

HEADERS:

Authorization: Bearer ⟨*JSONWebToken*⟩
Accept: application/json
Content-Type: application/json
Cache-Control: no-cache

DELETE /logout

```
[Body is empty, token is in Headers]
```

RESPONSE:

```
{
  "message":"Successfully logged out"
}
```

3.1.6 Create Virtual Silo

It starts creating a new Virtual Silo, unique for each tenant, with the characteristics and resources specified by the chosen flavour (default: *Mobius-base-f*). When this command is fired, if no error occured, a "silo is starting" message is returned. Once the command is completed and the Virtual Silo is created, information such as the private IP address of the vSilo broker, the port to be used for accessing it and the port mapping by which it is possible to access the broker using the public IP address of the platform, will become available. Progress and available information can be monitored by using the `/inspectVirtualSilo` endpoint of the API. It will fail if the indicated tenant does not match the authorized user.

Needed privileges: All users.

3.1.6.1 CLI

```
f4i.py create-vsilo [-h] [-c CONTROLLERURL] [-s VSILONAME] [-t TENANTID]
    [-f FLAVOURNAME] [-z VSILOZONE]
```

CLI Example

```
python3 f4i.py create-vsilo -c http://127.0.0.1:8090 -t tenant1 -f Mobius
    -base-f -s Silo1 -z default
```

3.1.6.2 REST API

HEADERS:

Authorization:	Bearer $\langle JSONWebToken \rangle$
Accept:	application/json
Content-Type:	application/json
Cache-Control:	no-cache

POST /siloCreate

```
{
  "tenantID":[TENANTID],
  "vSiloName":[VSILONAME],
  "flavourID":[FLAVOURNAME],
  "vSiloZone":[VSILOZONE],
  "debug_mode":false
}
```

RESPONSE:

```
{
  "message":"Virtual Silo is starting. Please wait..."
}
```

3.1.7 Destroy Virtual Silo

It starts deleting the Virtual Silo uniquely identified by its name and its tenant. The resources containing the Virtual Silo will be deleted, as well as any data. If the Virtual Silo had become unresponsive, the `force` or the corresponding CLI `-f` option can be used to perform a forced removal. When this command is fired, if no error occured, a "destroying virtual silo" message is returned. Progress and available information can be monitored by using the `/inspectVirtualSilo` endpoint of the API. It will fail if the indicated tenant does not match the authorized user.

Needed privileges: All users.

3.1.7.1 CLI

```
f4i.py destroy-vsilo [-h] [-c CONTROLLERURL] [-t TENANTID] [-s VSILONAME]
    [-f]
```

CLI Example

```
python3 f4i.py destroy-vsilo -c http://127.0.0.1:8090 -t tenant1 -s Silo1
```

3.1.7.2 REST API

HEADERS:

Authorization:	Bearer ⟨*JSONWebToken*⟩
Accept:	application/json
Content-Type:	application/json
Cache-Control:	no-cache

POST /siloDestroy

```
{
 "tenantID":[TENANTID],
 "vSiloName":[VSILONAME],
 "force":[false|true]
}
```

RESPONSE:

```
{
  "message":"Destroying virtual silo [TENANTID]_[VSILONAME]"
}
```

3.1.8 Add Virtual Thing

It adds a new Virtual Thing to the indicated Virtual Silo. It will fail if the Virtual Thing is not available in any ThingVisor of the platform, if the tenantID or vSiloID are not valid, if the Virtual Silo is not yet ready, or if the Virtual Thing already exists in the Virtual Silo. It will fail if the indicated tenant does not match the authorized user.

Needed privileges: All users.

3.1.8.1 CLI

```
python3 f4i.py add-vthing [-h] [-c CONTROLLERURL] [-t TENANTID] [-s
    VSILONAME] -v [VTHINGID]
```

CLI Example

```
python3 f4i.py add-vthing -c http://127.0.0.1:8090 -t tenant1 -s Silo1 -v
     weather/Tokyo_temp
```

3.1.8.2 REST API

HEADERS:

Authorization: Bearer ⟨*JSONWebToken*⟩
Accept: application/json
Content-Type: application/json
Cache-Control: no-cache

POST /addVThing

```
{
 "tenantID":[TENANTID],
 "vThingID":[VTHINGID],
 "vSiloName":[VSILONAME]
}
```

RESPONSE:

```
{
  "message":"vThing created"
}
```

3.1.9 Delete Virtual Thing

It deletes the Virtual Thing from the specified Virtual Silo of the indicated tenant. It will fail if the tenantID or vSiloID or vThingID are not valid. It will fail if the indicated tenant does not match the authorized user.

Needed privileges: All users.

3.1.9.1 CLI

```
f4i.py del-vthing [-h] [-c CONTROLLERURL] [-t TENANTID] [-s VSILONAME] [-
    v VTHINGID]
```

CLI Example

```
python3 f4i.py del-vthing -c http://127.0.0.1:8090 -t tenant1 -s Silo1 -v
     weather/Tokyo_temp
```

3.1.9.2 REST API

HEADERS:

Authorization:	Bearer ⟨*JSONWebToken*⟩
Accept:	application/json
Content-Type:	application/json
Cache-Control:	no-cache

POST /deleteVThing

```
{
 "tenantID":[TENANTID],
 "vThingID":[VTHINGID],
 "vSiloName":[VSILONAME]
}
```

RESPONSE:

```
{
  "message":"deleted virtual thing: [VTHINGID]"
}
```

3.1.10 List ThingVisors

All ThingVisors are listed. It gives back an array of JSON objects, each object giving information about a ThingVisor. In particular, it will dump all the details, including the parameters passed by the correponding add ThingVisor command, as well as the vThings associated to the ThingVisors.

Needed privileges: Administrator.

3.1.10.1 CLI

```
f4i.py list-thingvisors [-h] [-c CONTROLLERURL]
```

CLI Example

```
python3 f4i.py list-thingvisors -c http://127.0.0.1:8090
```

3.1.10.2 REST API

HEADERS:

Authorization: Bearer ⟨*JSONWebToken*⟩
Accept: application/json
Content-Type: application/json
Cache-Control: no-cache

GET /listThingVisors

```
[Body is empty, token is in Headers]
```

RESPONSE:

```
[
  {JSON object with information of a ThingVisor}
]
```

3.1.11 List Virtual Silos

Authorized user's Virtual Silos are listed (all Virtual Silos if user is the administrator). It gives back an array of JSON objects, each object giving information about a Virtual Silo.

Needed privileges: Administrator lists all Virtual Silos. Regular users can only list theirs.

3.1.11.1 CLI

```
f4i.py list-vsilos [-h] [-c CONTROLLERURL]
```

CLI Example

```
python3 f4i.py list-vsilos -c http://127.0.0.1:8090
```

3.1.11.2 REST API

HEADERS:

Authorization: Bearer ⟨*JSONWebToken*⟩
Accept: application/json
Content-Type: application/json
Cache-Control: no-cache

GET /listVirtualSilos

```
[Body is empty, token is in Headers]
```

RESPONSE:

```
[
  {JSON object with information of a Virtual Silo}
]
```

3.1.12 List Virtual Things

All Virtual Things are listed, regardless of the identity of the User making the API call. It gives back an array of JSON objects, each object giving information about a Virtual Thing.

Needed privileges: All users.

3.1.12.1 CLI

```
f4i.py list-vthings [-h] [-c CONTROLLERURL]
```

CLI Example

```
python3 f4i.py list-vthings -c http://127.0.0.1:8090
```

3.1.12.2 REST API

HEADERS:

Authorization: Bearer ⟨*JSONWebToken*⟩
Accept: application/json
Content-Type: application/json
Cache-Control: no-cache

GET /listVThings

```
[Body is empty, token is in Headers]
```

RESPONSE:

```
[
  {JSON object with information of a Virtual Thing}
]
```

3.1.13 Add ThingVisor

It starts adding a ThingVisor. The administrator can add a new ThingVisor from a docker image (using `imageName` or the corresponding CLI option `-i`) or from kubernetes yaml file(s) (using `yamlFiles` or the corresponding CLI option `-y`). In case the ThingVisor needs custom configuration parameters they are given as a JSON object by using `params` or the corresponding CLI option `-p`. When this command is fired, if no error occured, a "thingvisor is starting" message is returned. Once the command is completed and the ThingVisor is created, information such as the private IP address of the ThingVisor, the port to be used for accessing it and the port mapping by which it is possible to access it using the public IP address of the platform, will become available. Progress and available information can be monitored by using the `/inspectThingVisor` endpoint of the API.

Needed privileges: Administrator.

3.1.13.1 CLI

```
f4i.py add-thingvisor [-h] [-c CONTROLLERURL] [-i IMAGENAME]|[-y
    YAMLFILESPATH] [-n NAME] [-p PARAMS] [-d DESCRIPTION] [-z TVZONE] [--
    debug false|true]
```

CLI Example

```
python3 f4i.py add-thingvisor -c http://127.0.0.1:8090 -i fed4iot/v-
    weather-tv:2.2 -n weather -p "{'cities':['Rome', 'Tokyo','Murcia','
    Grasse','Heidelberg'], 'rate':60}" -d "Weather ThingVisor"
```

3.1.13.2 REST API

HEADERS: _____

Authorization: Bearer ⟨*JSONWebToken*⟩
Accept: application/json
Content-Type: application/json
Cache-Control: no-cache

POST /addThingVisor

```
{
  "imageName": [IMAGENAME],
  "thingVisorID": [NAME],
  "params": [PARAMS],
  "description": [DESCRIPTION],
  "debug_mode": [false|true],
  "tvZone": [TVZONE],
  "yamlFiles": [array of paths to yaml files]
}
```

RESPONSE:

```
{
  "message":"ThingVisor is starting. Please wait..."
}
```

3.1.14 Delete ThingVisor

It starts deleting a ThingVisor, also getting rid of any information about the specified ThingVisor in the System DB. If the ThingVisor had become unresponsive, `force` or the corresponding `-f` CLI option can be used to perform a forced removal. When this command is fired, if no error occured, a "deleting thingVisor" message is returned. Progress and available information can be monitored by using the `/inspectThingVisor` endpoint of the API.

Needed privileges: Administrator.

3.1.14.1 CLI

```
f4i.py del-thingvisor [-h] [-c CONTROLLERURL] [-n NAME] [-f]
```

CLI Example

```
python3 f4i.py del-thingvisor -c http://127.0.0.1:8090 -n weather
```

3.1.14.2 REST API

HEADERS:

Authorization: Bearer ⟨*JSONWebToken*⟩
Accept: application/json
Content-Type: application/json
Cache-Control: no-cache

POST /deleteThingVisor

```
{
  "thingVisorID":[NAME],
  "force":[false|true]
}
```

RESPONSE:

```
{
  "message":"deleting thingVisor: [NAME]"
}
```

3.1.15 Update ThingVisor

It starts updating a ThingVisor, changing information about the specified ThingVisor in the System DB. ThingVisor's configuration parameters, description and any ephemeral runtime information it needs, can be updated. Ephemeral information (by using `updateInfo` or the corresponding `-u` CLI option) is not saved into the system DB, it is passed to the ThingVisor, and its the usage depends on the specific ThingVisor implementation. When this command is fired, if no error occured, a "updating thingVisor" message is returned. Progress and available information can be monitored by using the `/inspectThingVisor` endpoint of the API.

Needed privileges: Administrator.

3.1.15.1 CLI

```
f4i.py update-thingvisor [-h] [-c CONTROLLERURL] [-n NAME] [-p PARAMS] [-
    d DESCRIPTION] [-u UPDATEINFO]
```

CLI Example

```
python3 f4i.py update-thingvisor -c http://127.0.0.1:8090 -n helloWorld -
    p '{"rate":1}' -d "Weather ThingVisor" -u "ephemeral information"
```

3.1.15.2 REST API

HEADERS:

Authorization:	Bearer ⟨*JSONWebToken*⟩
Accept:	application/json
Content-Type:	application/json
Cache-Control:	no-cache

POST /updateThingVisor

```
{
  "thingVisorID":[NAME],
  "params": [PARAMS],
  "description": [DESCRIPTION],
  "updateInfo": [UPDATEINFO]
}
```

RESPONSE:

```
{
  "message":"updating thingVisor: [NAME]"
}
```

3.1.16 Add Flavour

It starts adding a new Flavour to the platform. The administrator specifies the new Flavour ID as well as its parameters. The administrator can add a new Flavour from a docker image (using `imageName` or the corresponding CLI option `-i`) or from kubernetes yaml file(s) (using `yamlFiles` or the corresponding CLI option `-y`). In case the Flavour needs custom configuration parameters they are given as a JSON object by using `flavourParams` or the corresponding CLI option `-s`. This command immediately fails if the Flavour already exists or, in case of docker, if no corresponding image exists in the registered image repositories. When this command is fired, if no error occurred, a "flavour is being created" message is returned. Progress and available information can be monitored by using the `/inspectFlavour` endpoint of the API.

Needed privileges: Administrator.

3.1.16.1 CLI

```
f4i.py add-flavour [-h] [-c CONTROLLERURL] [-f FLAVOURID] [-s
    FLAVOURPARAMS] [-i IMAGENAME]|[-y YAMLFILESPATH] [-d DESCRIPTION]
```

CLI Example

```
python3 f4i.py add-flavour -c http://127.0.0.1:8090 -f Mobius-base-f -s
    Mobius -i fed4iot/mobius-base-f:2.2 -d "silo with a oneM2M Mobius
    broker"
```

3.1.16.2 REST API

HEADERS:

Authorization:	Bearer $\langle JSONWebToken \rangle$
Accept:	application/json
Content-Type:	application/json
Cache-Control:	no-cache

POST /addFlavour

```
{
  "flavourID":[FLAVOURID],
  "flavourParams":[FLAVOURPARAMS],
  "imageName":[IMAGENAME],
  "flavourDescription":[DESCRIPTION],
  "yamlFiles": [array of paths to yaml files]
}
```

RESPONSE:

```
{
  "message":"Flavour is being created. Please wait..."
}
```

3.1.17 Delete Flavour

It deletes a Flavour, removing it from the System DB.

Needed privileges: Administrator.

3.1.17.1 CLI

```
f4i.py del-flavour [-h] [-c CONTROLLERURL] [-n FLAVOURID]
```

CLI Example

```
python3 f4i.py del-flavour -c http://127.0.0.1:8090 -n Mobius-base-f
```

3.1.17.2 REST API

HEADERS:

Authorization:	Bearer ⟨*JSONWebToken*⟩
Accept:	application/json
Content-Type:	application/json
Cache-Control:	no-cache

POST /deleteFlavour

```
{
  "flavourID":[FLAVOURID]
}
```

RESPONSE:

```
{
  "message":"Flavour [FLAVOURID] deleted"
}
```

3.1.18 List Flavours

It lists all the Flavours available in the platform.

Needed privileges: All users.

3.1.18.1 CLI

```
f4i.py list-flavours [-h] [-c CONTROLLERURL]
```

CLI Example

```
python3 f4i.py list-flavours -c http://127.0.0.1:8090
```

3.1.18.2 REST API

HEADERS:

Authorization:	Bearer ⟨*JSONWebToken*⟩
Accept:	application/json
Content-Type:	application/json
Cache-Control:	no-cache

GET /listFlavours

```
[Body is empty, token is in Headers]
```

RESPONSE:

```
[
  {JSON object with information of a Flavour}
]
```

3.1.19 Inspect Tenant

It dumps information about the indicated Tenant ID. In particular, it will print all the information about the Virtual Silos associated to the tenant (and their respective Virtual Things). It will fail if the indicated tenant does not match the authorized user.

Needed privileges: All users.

3.1.19.1 CLI

```
f4i.py inspect-tenant [-h] [-c CONTROLLERURL] [-t TENANTID]
```

CLI Example

```
python3 f4i.py inspect-tenant -c http://127.0.0.1:8090 -t tenant25
```

3.1.19.2 REST API

HEADERS:

Authorization: Bearer ⟨*JSONWebToken*⟩
Accept: application/json
Content-Type: application/json
Cache-Control: no-cache

POST /inspectTenant

```
{
  "tenantID":[TENANTID]
}
```

RESPONSE:

```
{
  "vSilos":[descriptions of tenant's vsilos],
  "vThings":[descriptions of tenant's vthings]
}
```

3.1.20 Inspect Virtual Silo

It prints information about the indicated Virtual Silo. In particular, it will print the description of the Virtual Silo as well as its respective Virtual Things. It will fail if the authorized user does not own the indicated Virtual Silo.

Needed privileges: All users.

3.1.20.1 CLI

```
f4i.py inspect-vsilo [-h] [-c CONTROLLERURL] [-v VSILOID]
```

CLI Example

```
python3 f4i.py inspect-vsilo -c http://127.0.0.1:8090 -v tenant1_Silo1
```

3.1.20.2 REST API

HEADERS:

Authorization: Bearer ⟨*JSONWebToken*⟩
Accept: application/json
Content-Type: application/json
Cache-Control: no-cache

POST /inspectVirtualSilo

```
{
  "vSiloID":[VSILOID]
}
```

RESPONSE:

```
{
  "vSilo":[description of vsilo],
  "vThings":[description of vsilos's vthings]
}
```

3.1.21 Inspect ThingVisor

It gives back a description of the requested ThingVisor ID.
Needed privileges: Administrator.

3.1.21.1 CLI

```
f4i.py inspect-thingvisor [-h] [-c CONTROLLERURL] [-v THINGVISORID]
```

CLI Example

```
python3 f4i.py inspect-thingvisor -c http://127.0.0.1:8090 -v weather
```

3.1.21.2 REST API

HEADERS:

Authorization: Bearer ⟨*JSONWebToken*⟩
Accept: application/json
Content-Type: application/json
Cache-Control: no-cache

POST /inspectThingVisor

```
{
  "thingVisorID":[THINGVISORID]
}
```

RESPONSE:

```
[
  {JSON object describing the ThingVisor's details}
]
```

3.1.22 Inspect Flavour

It gives back a description of the requested Flavour ID.
Needed privileges: Administrator.

3.1.22.1 CLI

```
f4i.py inspect-flavour [-h] [-c CONTROLLERURL] [-v FLAVOURID]
```

CLI Example

```
python3 f4i.py inspect-flavour -c http://127.0.0.1:8090 -v Mobius-base-f
```

3.1.22.2 REST API

HEADERS:

Authorization: Bearer ⟨*JSONWebToken*⟩
Accept: application/json
Content-Type: application/json
Cache-Control: no-cache

POST /inspectFlavour

```
{
  "flavourID":[FLAVOURID]
}
```

RESPONSE:

```
[
  {JSON object describing the Flavour's details}
]
```

3.1.23 Set Virtual Thing Endpoint

Used by administrator to set the HTTP endpoint for a Virtual Thing, to be used for communications to and from the Virtual Thing via the HTTP data distribution system. It tells the Virtual Thing to enable the requested endpoint, and it creates a corresponding entry for it in the System DB. It fails if the Virtual Thing does not exist or the ThingVisor hosting the Virtual Thing is not ready yet.

Needed privileges: Administrator.

3.1.23.1 CLI

```
f4i.py set-vthing-endpoint [-h] [-c CONTROLLERURL] [-v VTHINGID] [-e
    ENDPOINT]
```

CLI Example

```
python3 f4i.py set-vthing-endpoint -c http://127.0.0.1:8090 -v relay-tv-
    jp/timestamp -e http://127.0.0.1:8081
```

3.1.23.2 REST API

HEADERS:

Authorization:	Bearer ⟨*JSONWebToken*⟩
Accept:	application/json
Content-Type:	application/json
Cache-Control:	no-cache

POST /setVThingEndpoint

```
{
  "vThingID":[VTHINGID],
  "endpoint":[ENDPOINT]
}
```

RESPONSE:

```
{
  "message":"vThing endpoint created"
}
```

3.1.24 Delete Virtual Thing Endpoint

Used by administrator to remove the HTTP endpoint for a Virtual Thing. It tells the Virtual Thing to disable its HTTP endpoint, and it removes the corresponding entry in the System DB. It fails if the Virtual Thing does not exist or the ThingVisor hosting the Virtual Thing is not ready yet.

Needed privileges: Administrator.

3.1.24.1 CLI

```
f4i.py del-vthing-endpoint [-h] [-c CONTROLLERURL] [-v VTHINGID]
```

CLI Example

```
python3 f4i.py del-vthing-endpoint -c http://127.0.0.1:8090 -v relay-tv-
    jp/timestamp
```

3.1.24.2 REST API

HEADERS:

Authorization:	Bearer ⟨*JSONWebToken*⟩
Accept:	application/json
Content-Type:	application/json
Cache-Control:	no-cache

POST /delVThingEndpoint

```
{
  "vThingID":[VTHINGID]
}
```

RESPONSE:

```
{
  "message":"vThing endpoint deleted"
}
```

3.2 VirIoT Control Procedures

This section describes the procedures and the related internal signaling that support both the management operations described in section 3.1 and the internal data flows that deliver the data.

Although VirIoT can run using Docker on a single machine (as presented in D3.1), the latest architecture (see D2.3) uses Kubernetes, as a Compute Virtualization Layer, to enable efficient distributed deployments on infrastructures formed by many cloud/edge computing sites. Consequently, the procedures will be described considering a Kubernetes-based VirIoT deployment.

3.2.1 VirIoT Control Commands

VirIoT services exchange control messages to control their configuration. As described in D4.2 these messages use the MQTT distribution system. Table 3 describes the identifiers used in control messages. Table 4 describes the control messages and their relationship to MQTT in terms of who are the publishers, the subscribers, and what topics are used for transfer in the network. The MQTT message body of control messages is shown in Table 5. The following sections describe VirIoT procedures that use these control messages.

Identifier	Type	Format	Description
tenantID	string	DNS subdomain name	Unique identifier of a tenant in the system
vSiloName	string	DNS subdomain name	Unique identifier of the vSilo among those of a tenant
vSiloID	string	<tenantID>_<vSiloName>	Unique identifier of the vSilo in the system
thingVisorID	string	DNS subdomain name	Unique identifier of a ThingVisor in the system
vThingName	string	DNS subdomain name	Unique identifier of the vThing among those of a ThingVisor
vThingID	string	<ThingVisorID>/<vThingName>	Unique identifier of a vThing in the system

Table 3: VirIoT identifiers

Command	Publisher	Subscriber	MQTT Topic	Description
addVThing	Master Controller	vSilo controllers	vSilo/<vSiloID>/c_in	Connect a vThing to the vSilo whose ID is vSiloID

deleteVThing	Master Controller ThingVisor	vSilo controllers	vSilo/<vSiloID>/c_in vThing/<vThingID>/c_out	Disconnect a vThing from the vSilo whose ID is vSiloID. Can be issued by Master Controller towards a specific vSiloID as a consequence of a request from the tenant of the vSilo to remove the vThing. Can be issued by a ThingVisor to remove the vThing from all connected vSilos
createVThing	ThingVisor	Master Controller	TV/<thingVisorID>/c_out	A ThingVisor whose identifier is thingVisorID, notifies the presence of a new vThing, whose identifier is vThingID.
destroyTV	Master Controller	ThingVisor	TV/<thingVisorID>/c_in	Remove a ThingVisor from the system whose identifier is thingVisorID
destroyTVAck	ThingVisor	Master Controller	TV/<thingVisorID>/c_out	A ThingVisor, whose identifier is thingVisorID, notifies that it is ready to be removed from the system. Subsequently, Master Controller destroys its instance (container/POD)
destroyVSilo	Master Controller	vSilo controllers	vSilo/<vSiloID>/c_in	Remove a vSilo from the system whose identifier is vSiloID
destroyVSiloAck	IoT vSilo controller	Master Controller	vSilo/<vSiloID>/c_out	A vSilo, whose identifier is vSiloID, notifies that it is ready to be removed from the system. Subsequently, Master Controller destroys its instance (container/POD)
getContextRequest	IoT vSilo controller	ThingVisor	vThing/<vThingID>/c_in	Get the current context data of a vThing whose identifier is vThingID
getContextResponse	ThingVisor	IoT vSilo controller	vSilo/<vSiloID>/c_in	Response to getContextRequest message
updateTV	Master Controller	ThingVisor	TV/<thingVisorID>/c_in	Update possible parameters of a ThingVisor
setVThingEndpoint	Master Controller	ThingVisor	TV/<thingVisorID>/c_in	Configure an HTTP external endpoint where to forward HTTP requests issued by vSilo and related to generic contents of a vThing whose ThingVisor identifier is thingVisorID. This command is used by the ThingVisor HTTP Sidecar container (that can be added to the Kubernetes POD of an existing ThingVisor) to enable management of generic HTTP contents
delVThingEndpoint	Master Controller	ThingVisor	TV/<thingVisorID>/c_in	Remove the an HTTP external endpoint of a vThing

Table 4: Control commands exchanged among VirIoT services

Command	JSON MQTT body

addVThing	{
	"command": "addVThing"
	"vSiloID": <vSiloID>,
	"vThingID": <vThingID>,
	"vThingType": string
	}

deleteVThing	{
	"command": "deleteVThing",
	"vSiloID": <vSiloID> or "ALL" to refer to any vSilo,
	"vThingID": <vThingID>
	}

createVThing	{
	"command": "createVThing",
	"thingVisorID": <thingVisorID>,
	"vThing":{"label": string, "id": <vThingID>, "description": string, "type": string}
	}

destroyTV	{
	"command": "destroyTV",
	"thingVisorID": <thingVisorID>
	}

destroyTVAck	{
	"command": "destroyTVAck",
	"thingVisorID": <thingVisorID>
	}

destroyVSilo	{
	"command": "destroyVSilo",
	"vSiloID": <vSiloID>
	}

destroyVSiloAck	{
	"command": "destroyVSiloAck",
	"vSiloID": <vSiloID>
	}

getContextRequest	{
	"command": "getContextRequest",
	"vSiloID": <vSiloID>,
	"vThingID": <vThingID>
	}

getContextResponse	{
	"command": "getContextResponse",
	"data": list of NGSI-LD entities,
	"meta": {"vThingID": <vThingID>}
	}

updateTV	{
	"command": "updateTV",
	"thingVisorID": <thingVisorID>,
	"tvDescription": string,
	"params": json object,
	"updateInfo": string
	}

setVThingEndpoint	{
	"command": "setVThingEndpoint",
	"vThingID": <vThingID>,
	"endpoint": URL of endpoint
	}

delVThingEndpoint	{
	"command": "setVThingEndpoint",
	"vThingID": <vThingID>
	}

Table 5: MQTT JSON body of the control commands exchanged among VirIoT services

3.2.2 User management procedures

Figure 5 shows the register procedure used to insert a new user in the VirIoT system. At the bottom of each figure, we will show the protocols used among the involved entities. In the figures we included some details about the content exchanged through API and control messages. However, the full description of this information can be found in the previous sections on API and control messages. In the next figures we assume two possible

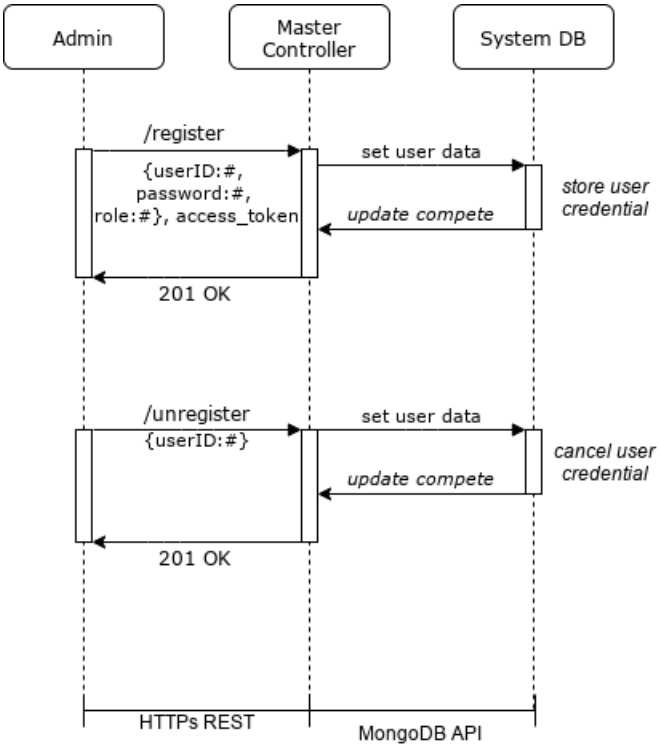


Figure 5: Register/Unregister procedure

roles for a user: Administrator or Tenant.

During the register procedure the Admin sends a register request (HTTPs POST) to the Master Controller using the `/register` REST resource, including JSON information such as `userID`, `password`, `role` that are related to the user the Admin is registering. Moreover, the Admin sends her `access_token` (a JSON Web Token - JWT) in the Auth header of the HTTP POST. In this and next figures, we use curly brackets to indicate JSON content and we will use the symbol # to indicate a generic value of the key. The Master Controller verifies the validity of the access token and consequently stores user information in the System DB, currently MongoDB. Only the HASH of the password is actually stored in the DB, so that only the user knows her real password. Figure 5 also shows the unregister procedure that is very similar, except that here the Master Controller removes user information from the System DB.

Figure 6 shows the login procedure that is used by a user (e.g. a tenant) to get her `access_token`. The user sends her credentials through HTTPs to the REST `/login` resource. The Master Controller accesses the System DB to fetch user info, then verifies the password HASH and, if the verification is successful, it sends back to the user her `access_token` that will be used within subsequent procedures for authentication purposes. Figure 6 also shows the logout procedure in which the `access_token` of the user is inserted in a local black-list, thus it can no longer be used.

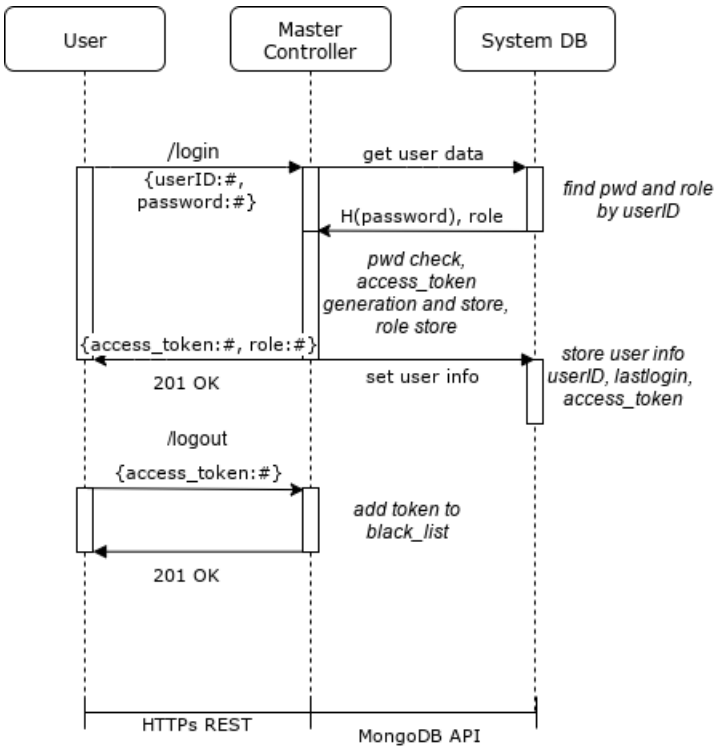


Figure 6: Login/Logout procedure

3.2.3 Virtual Silo procedures

This section reports procedures related to the creation and removal of a Virtual Silo (vSilo). We remind that a vSilo starts as an instance of a "void" Virtual Silo image that we named *Flavour*, which is a docker image or a set of them packaged in a Kubernetes POD described by a YAML file. Beside Docker, other lighter container run-times have been studied as reported in section 5.

Figure 7 shows the procedure used by the Admin to add a Flavour to the VirIoT system. The Admin uses the `/addFlavour` REST resource to transfer information of the Flavour, such as description, image name (a URL), flavour identifier, YAML file describing the vSilo, etc. The Master Controller registers this information in the System DB, by setting the Flavour status as PENDING. Immediately, it sends back an HTTP 201 OK to the Tenant to quickly release the connection. As previously introduced, we used such "transient" statuses in many other procedures to quickly disconnect the HTTP client (aka "fire and forget" approach), hence avoiding it to stay connected for many seconds, especially for those procedures that require long backend operations, such as the download of a Docker image. Thereafter, the Master Controller checks the availability of the image on the registered image repositories (e.g. DockerHub or local repo); if the image exists, the Master Controller changes the status to READY, otherwise, it cancels the pending entry in the System DB. Figure 7 also shows the delete Flavour procedure in which the Master Controller removes Flavour's data from the System DB.

Figure 8 shows how a Tenant can create a Virtual Silo (vSilo). The Tenant sends to the Master Controller some information, such as the identifier of the vSilo flavour to be

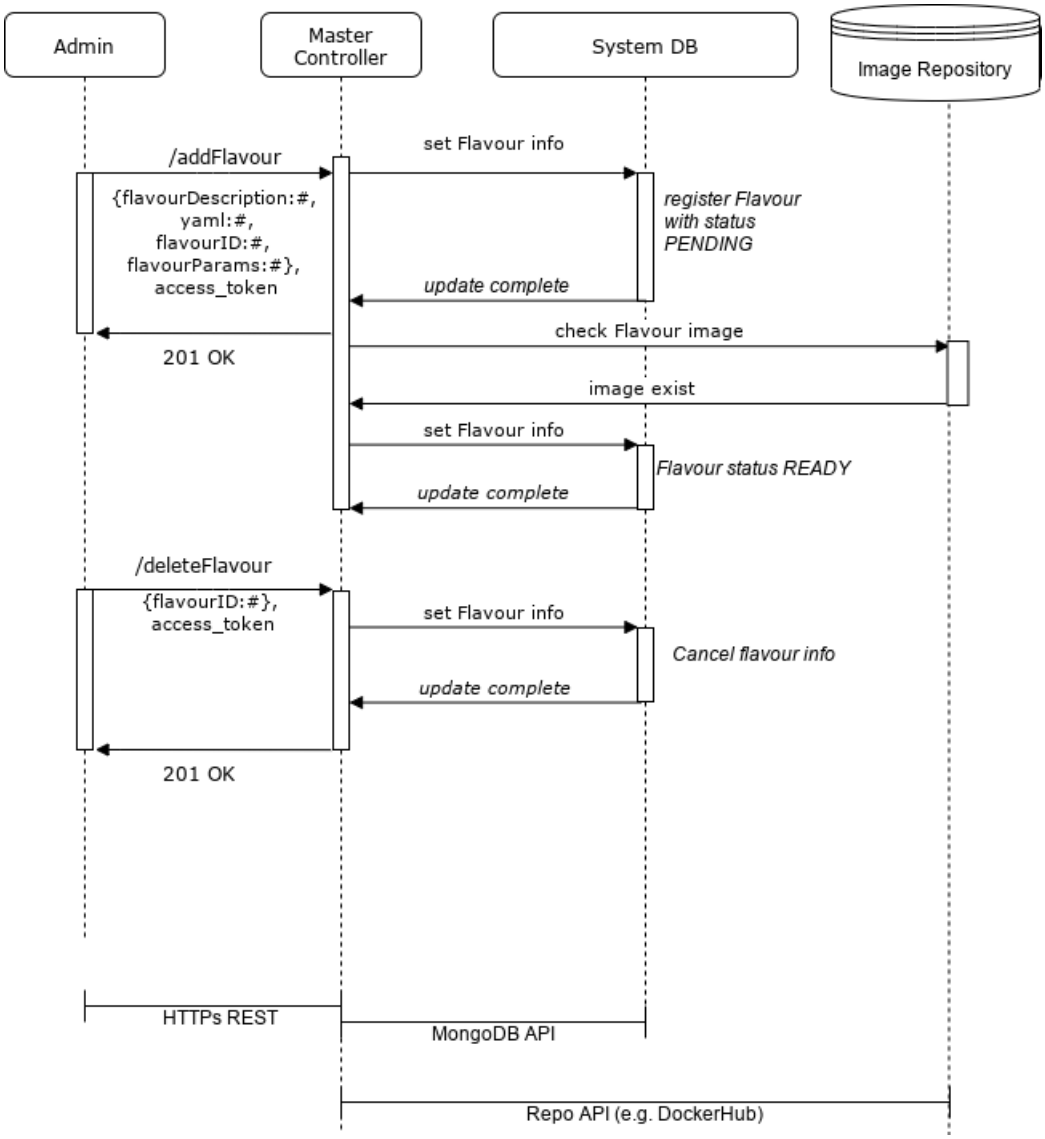


Figure 7: Add Flavour procedure

used as the base image and the name she wishes to use for the Virtual Silo. The Master Controller registers the information of the new Virtual Silo in the System DB and sets the status as PENDING. Thereafter, it requests to the Compute Virtualization Layer, e.g. Kubernetes (K8S), to deploy the instance of the vSilo. The Compute Virtualization Layer downloads the related image (Flavour) from the Repository and runs a new instance of it. When the deployment is complete the Master Controller stores in the System DB the vSilo configuration information, such as the private IP address, the exposed port, the public IP address of the gateway of the VirIoT zone, the instance ID (e.g. container/pod name), etc. and sets vSilo status as RUNNING.

Figure 9 shows how a Tenant can destroy a Virtual Silo (vSilo). The Tenant sends to the Master Controller information such as the identifier of the vSilo to be destroyed. The Master Controller gets the vSilo information from the System DB, sets the status as STOPPING and replies to the user with an HTTP 201 OK. Then, the Master Controller

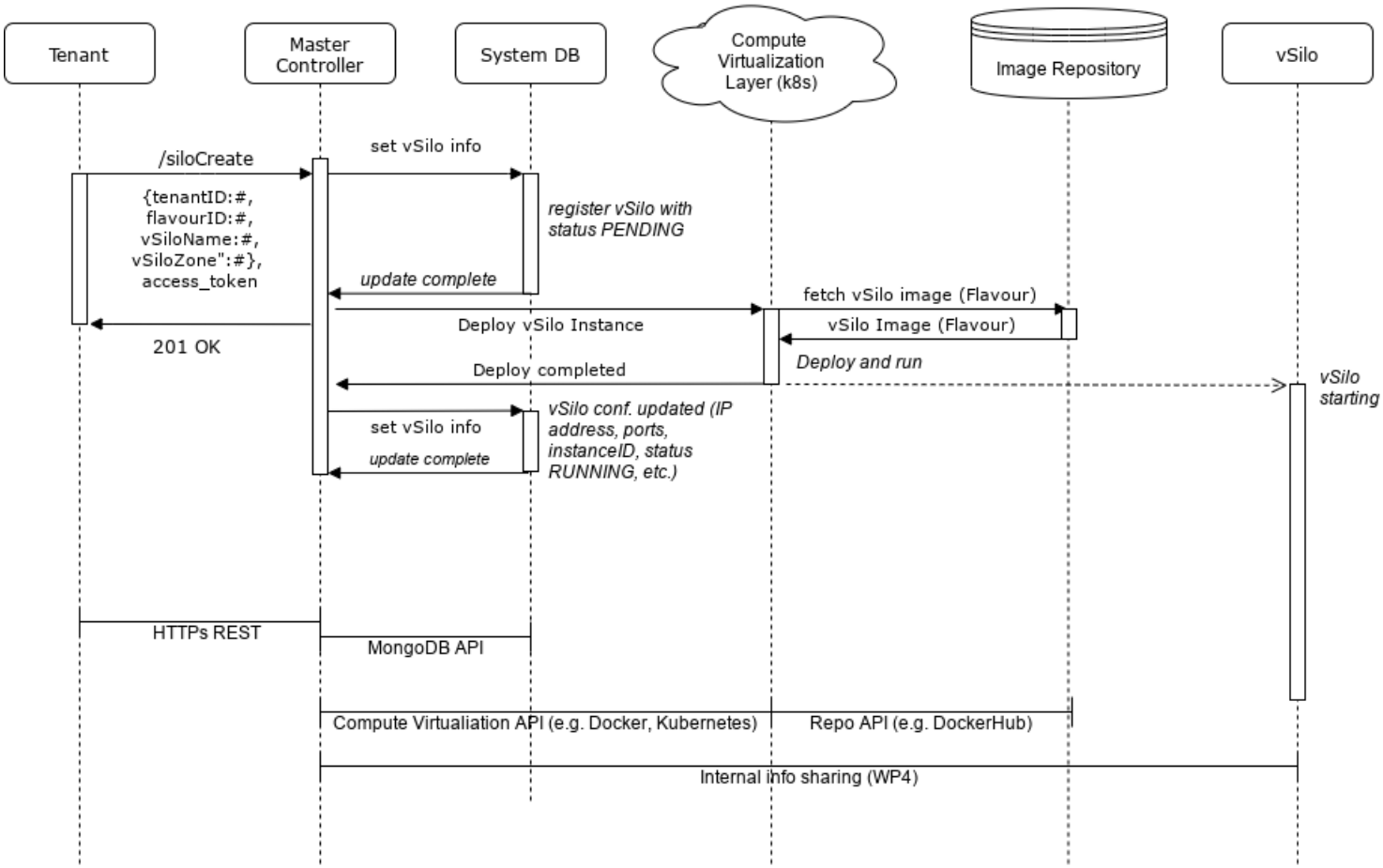


Figure 8: Create vSilo procedure

asks the vSilo Controller to start the destroy procedure. This message is a VirIoT *control command* delivered through the MQTT cluster of the Internal Info Sharing VirIoT subsystem (see D4.2).

When the vSilo Controllers (IoT and HTTP) receive the `destroyVSilo` command they gracefully close all possible relations with external sources and then the IoT controller confirms to the Master Controller that it is ready to be destroyed with a `destroyVSiloAck` command. At the reception of this message, the Master Controller requests to the Compute Virtualization Layer to destroy the vSilo instance. When the destroy operation is completed, the Master Controller removes vSilo information from the System DB.

Figure 10 is the Kubernetes YAML files associated to a oneM2M/Mobius vSilo flavour. The YAML in Figure 10 shows what practically is an instance of a vSilo. A vSilo is a Kubernetes Deployment made of a single POD (replicas = 1). The POD contains two containers:

- a container (i.e. fed4iot/mobius2-pub-sub-actuator-f) that includes the IoT Controller and the Broker of the vSilo;
- a container (i.e. fed4iot/http-sidecar-flavour) that includes the HTTP Controller and Broker of the vSilo

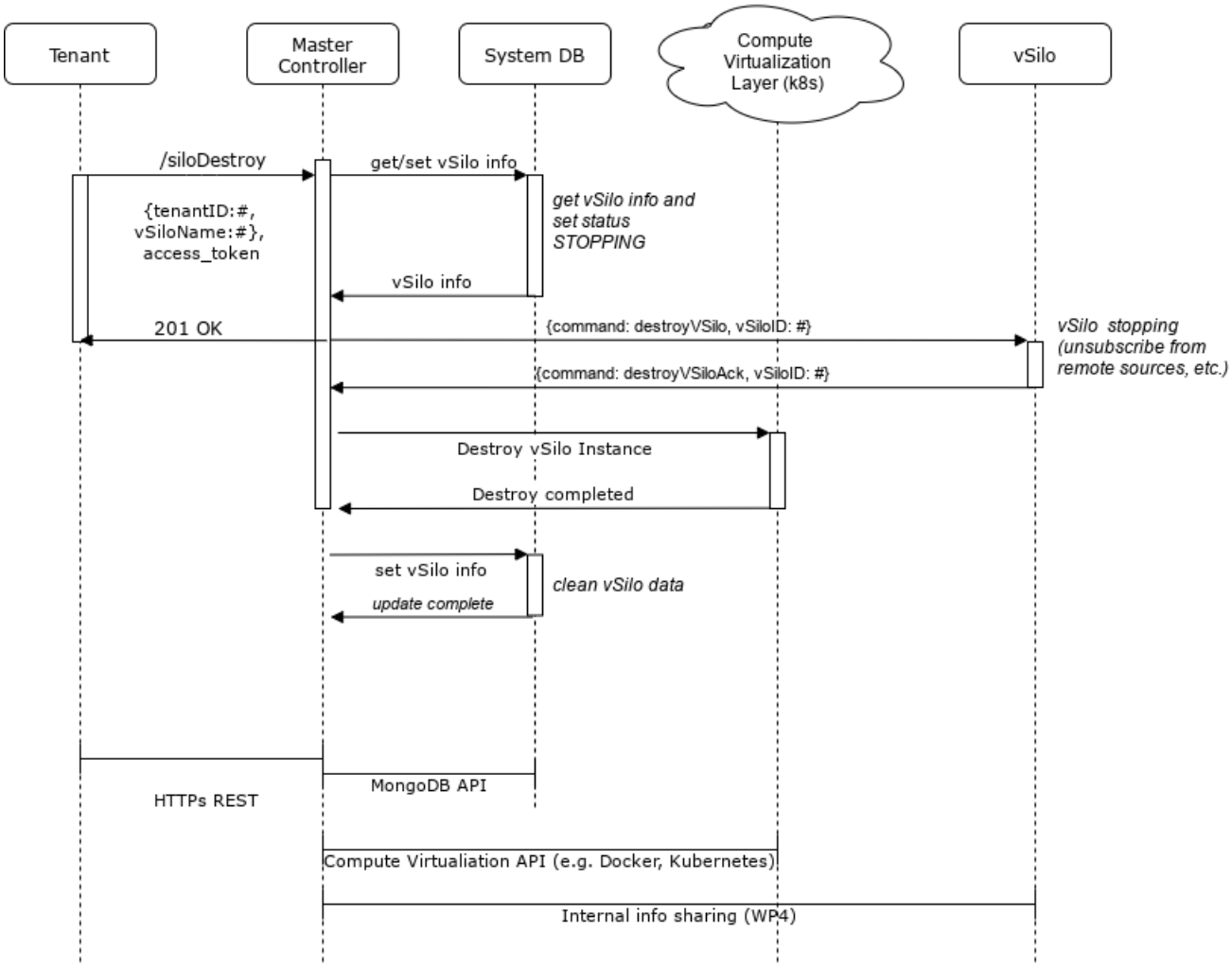


Figure 9: Destroy vSilo procedure

A Kubernetes NodePort service is associated to the vSilo POD, to enable a tenant to contact her vSilo through the IP address of the gateway node of the VirIoT zone where it is deployed.

Such a YAML file is used by the Master Controller to create a vSilo. During the creating phase, the Master Controller uses this *generic* file to create a vSilo-specific YAML file that is used to eventually run the vSilo (see Figure 67). The resulting vSilo YAML file also includes nodeAffinity constraints that enable Kubernetes to control the VirIoT Zone where to deploy the vSilo as specified in the API/CLI, as we will discuss in Section 5.1. Moreover, it also includes minimal environment variables necessary to the vSilo Controllers to contact the System DB and the MQTT Distribution System.

3.2.4 ThingVisor procedures

Figure 11 shows the procedure used by the Admin to add a ThingVisor to the VirIoT system. The Admin contacts the Master Controller through the `/addThingVisor` REST resource and sends ThingVisor information such as a description, its YAML file, the

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: f4i-mobius-pub-sub-actuator-f
spec:
  replicas: 1
  selector:
    matchLabels:
      app: f4i-mobius-pub-sub-actuator
  template:
    metadata:
      labels:
        app: f4i-mobius-pub-sub-actuator
    spec:
      containers:
      - name: f4i-mobius-pub-sub-actuator-f
        image: fed4iot/mobius2-pub-sub-actuator-flavour:latest
        ports:
        - containerPort: 1883
        - containerPort: 7579
      - name: http-sidecar-f
        image: fed4iot/http-sidecar-flavour:latest
        ports:
        - containerPort: 5001
---
apiVersion: v1
kind: Service
metadata:
  name: f4i-mobius-pub-sub-actuator-svc
  labels:
    app: f4i-mobius-pub-sub-actuator
spec:
  type: NodePort
  selector:
    app: f4i-mobius-pub-sub-actuator
  ports:
  - port: 7579
    nodePort:
    name: mobius-http
  - port: 1883
    nodePort:
    name: mobius-mqtt
  - port: 80
    targetPort: 5001
    nodePort:
    name: http
```

Figure 10: Example of a Kubernetes YAML file of a oneM2M vSilo

parameters that can be used to customize the added ThingVisor (e.g. info about the real sources that should be contacted by the ThingVisor), etc. The Master Controller registers ThingVisor information in the System DB, sets the status as PENDING, then sends back an HTTP 201 OK to the Admin. Thereafter, the Master Controller asks the Compute Virtualization Layer to deploy an instance of the ThingVisor. The Virtualization Layer fetches the image from the repository and runs it. In turn, the Master Controller updates the ThingVisor information in the System DB by setting its status to RUNNING. A ThingVisor may manage more than one Virtual Things (vThings), which are not known a-priori by the Master Controller. For this reason, when the ThingVisor starts, it sends to the Master Controller the `createVThing` command, which includes the list of vThings it is handling. At the reception of this message, the Master Controller updates the System DB with vThings' information.

Figure 12 shows the procedure used by a Tenant to connect (add) a vThing to her vSilo. The Tenant sends vThing and vSilo information to the Master Controller by using the `/addVThing` REST resource. The Master Controller checks the existence of vThing and vSilo, and then send the `addVThing` control command to the Controllers of the vSilo.

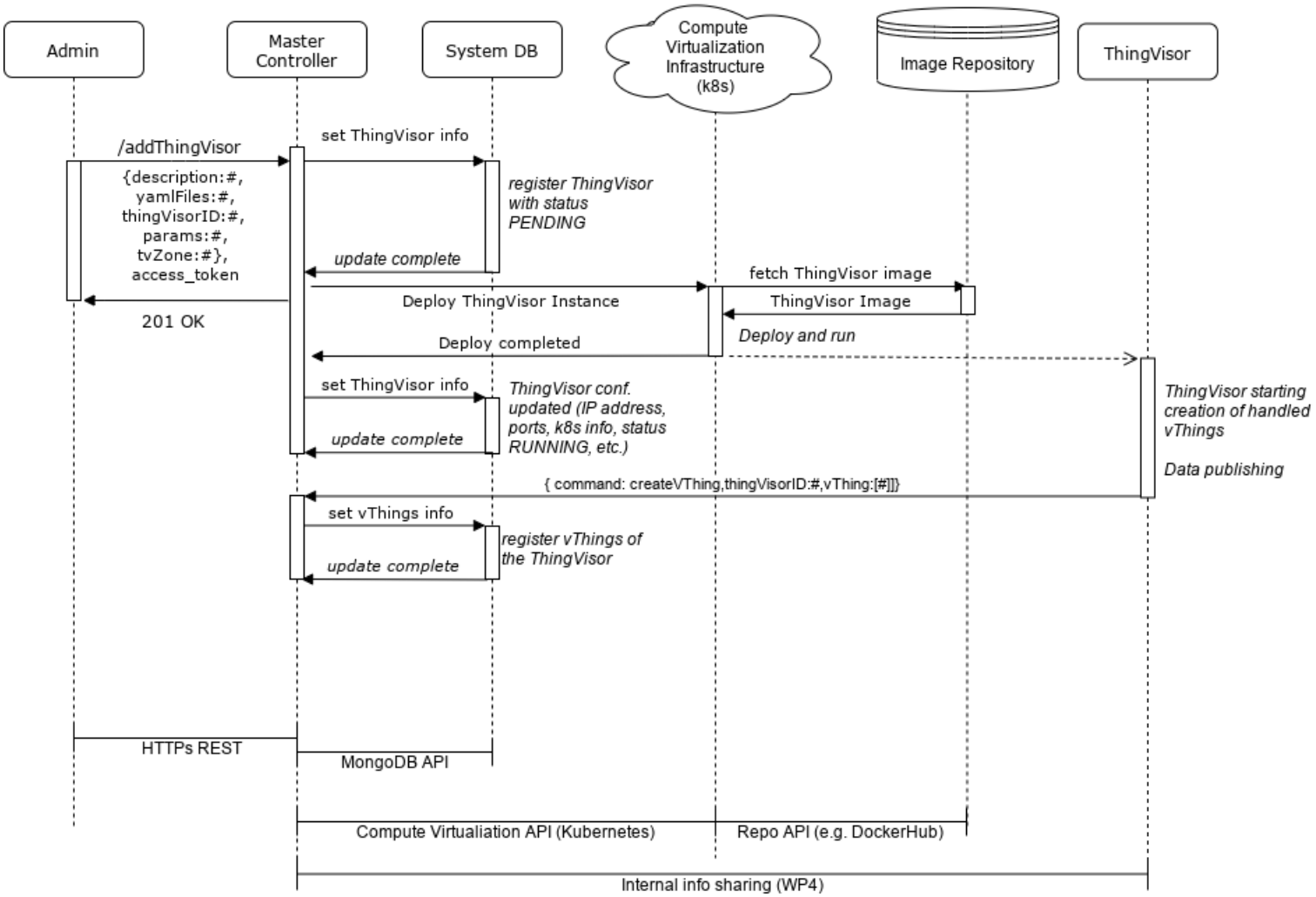


Figure 11: Add ThingVisor procedure

In turn,

- the IoT Controller subscribes the vThing control/data out topics so as to receive control messages and context data from the vThing; requests the latest context data published by the ThingVisor by using the `getContexRequest` control command.
- the HTTP Controller enables the forwarding of HTTP GET/POST towards the HTTP endpoint of the vThing, namely `http://vSiloIP:port/vstream/<vThingID>`

Meanwhile, the Master Controller updates the vSilo information on the System DB.

The MQTT Distribution System then starts to deliver context data items (in NGSI-LD format) to the vSilos. When a context data piece is received, the IoT Controller translate it to the information model used by the IoT Broker of the vSilo (see D4.2) and then inserts this information into the IoT Broker.

A tenant can fetch context data from the vSilo Broker and can carry out a HTTP GET/POST request through the vSilo HTTP. The (external) URL prefix used by the tenant for this request is `http://vSiloIP:port/vstream/<ThingVisorID>/<vThingName>`

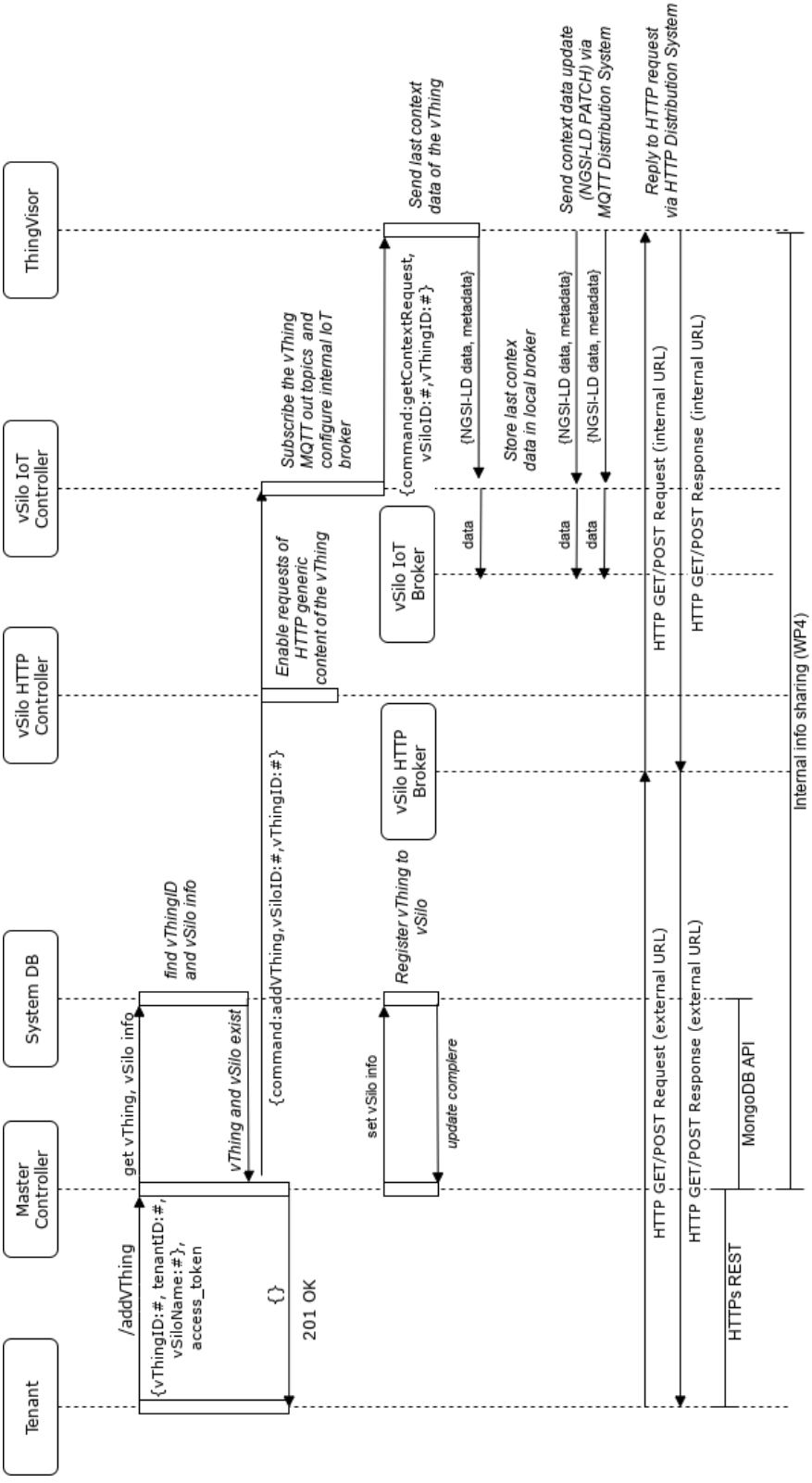


Figure 12: Add vThing procedure

`/<suffix-path>`. Such a request is received by the HTTP Broker and translated by it to a (internal) URL routed by the HTTP Distribution System to the ThingVisor. The internal URL has the form `http://<ThingVisor-K8S-service-name>/<vThingName>/<suffix-path>`.

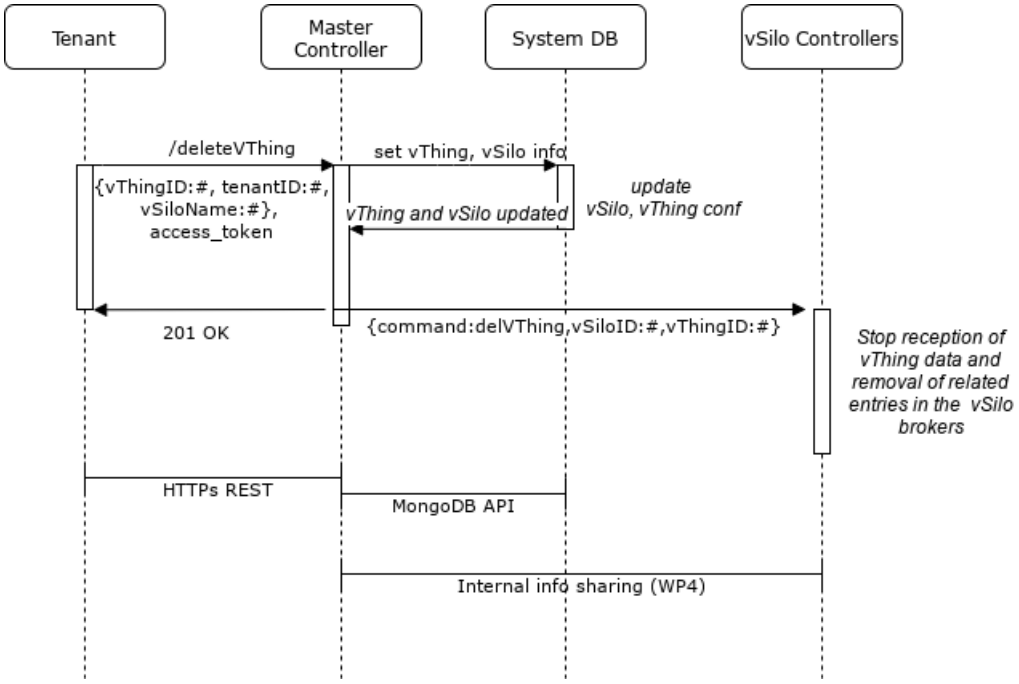


Figure 13: Delete vThing procedure

Figure 13 shows the procedure used by a Tenant to disconnect (delete) a vThing from her vSilo. The Tenant sends vThing and vSilo information to the Master Controller by using the `/deleteVThing` REST resource. The Master Controller checks the existence of vThing and vSilo, updates System DB, hence removing the vThing from the vSilo and then sends the `delVThing` control command to the Controllers of the vSilo. The Controllers stops receiving vThing data and removes related entries from the IoT Broker.

Figure 14 shows the procedure to remove a ThingVisor from the VirIoT system. The Admin contacts the Master Controller by using the `/deleteThingVisor` REST resource and passing the related ThingVisor ID. The Master Controller gets from the System DB information about the vSilos that are connected to the vThings handled by the ThingVisor, and sets the ThingVisor status to STOPPING. Then, the Master Controller sends `delVThing` control commands to the interested vSilos, so that they can remove the vThings of the ThingVisor from their brokers and stop receiving related data. Afterwards, the Master Controller asks the ThingVisor to stop itself by using the `destroyTV` control command. At the reception of this command, the ThingVisor revokes possible external states (e.g. subscriptions with remote sources, etc.) and then confirms that it ready to be destroyed by sending back the `destroyTVAck` control message to the Master Controller. When received, the Master Controller eventually asks the Compute Virtualization Layer to remove the ThingVisor instance.

Figure 15 shows the procedure used by the the Administrator to update the parameters of a running ThingVisor. Finally, Figure 16 and Figure 17 show the procedures used

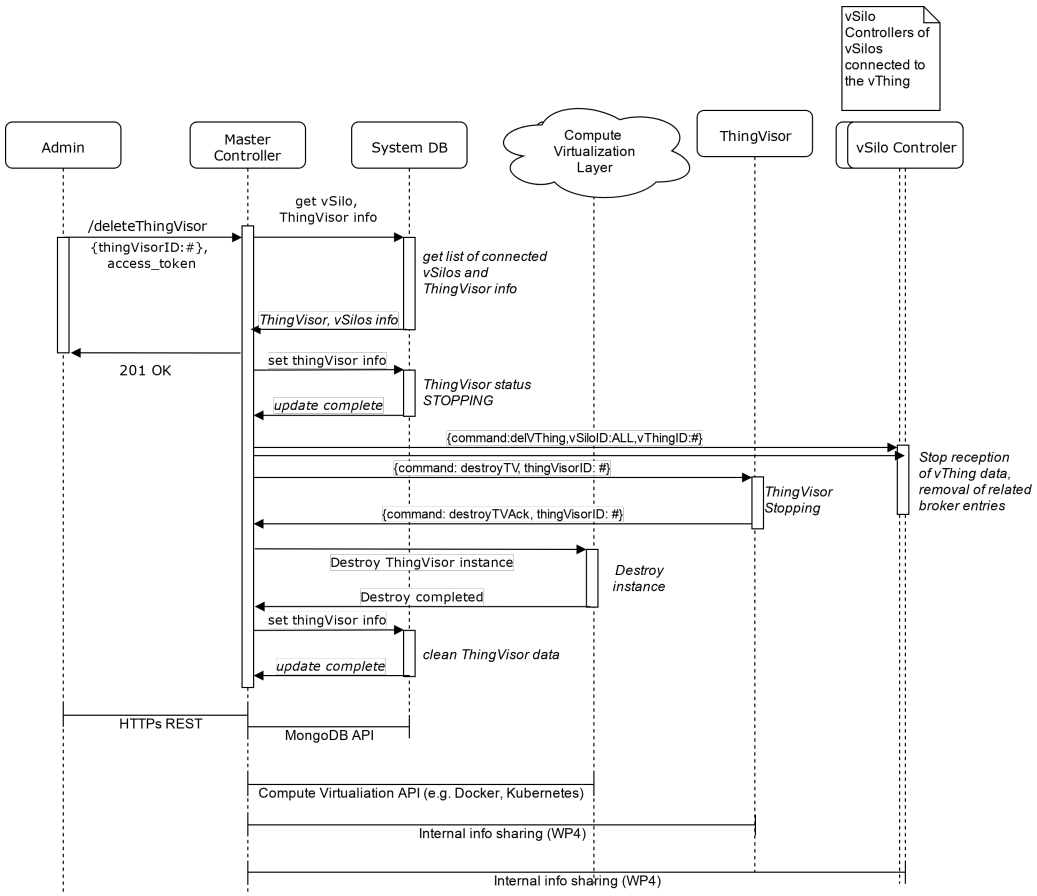


Figure 14: Delete ThingVisor procedure

to configure and delete a vThing HTTP endpoint (see also Section 3.3.1).

Figure 18 is the Kubernetes YAML files associated to a ThingVisor named Relay ThingVisor. The YAML in Figure 18 shows what practically is an instance of a ThingVisor. A ThingVisor is a Kubernetes Deployment made of a single POD (replicas = 1). The POD contains two containers:

- a container (i.e. fed4iot/relay-tv) that includes the ThingVisor software that handles context data;
- a container (i.e. fed4iot/http-sidecar-tv) that includes a generic software which handles HTTP generic contents (Section 3.3.1).

A Kubernetes NodePort service is associated to the ThingVisor POD, to enable external real things to contact the ThingVisor through the IP address of the gateway node of the VirIoT zone where it is deployed.

Such a YAML file is used by the Master Controller to create a ThingVisor. During the creating phase, the Master Controller uses this *generic* file to create a ThingVisor-specific YAML file that is used to eventually run the ThingVisor. The resulting ThingVisor YAML file also includes affinity constraints that make possible to exploit Kubernetes to control the VirIoT Zone where to deploy the ThingVisor as specified in the API/CLI.

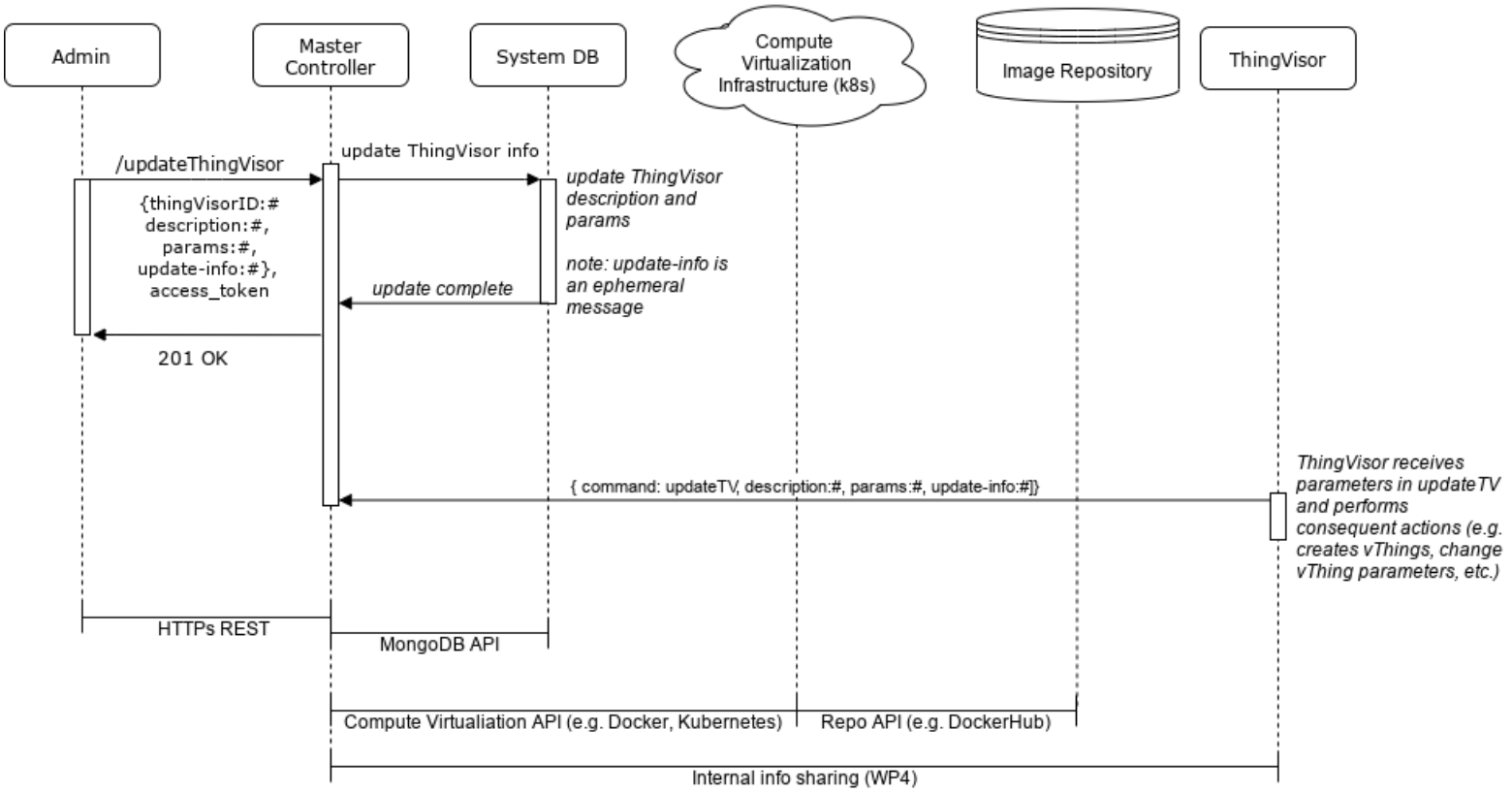


Figure 15: Update ThingVisor procedure

Moreover, it also includes minimal environment variables necessary to the ThingVisor to contact the System DB and the MQTT Distribution System.

3.2.5 System DB

The System DB stores the run-time configuration of VirIoT. Currently, the database in use is MongoDB. The data is organized in collections, which are (to some extent) analogous to tables in relational databases. A collection stores documents, which can be different in structure, and this is possible since MongoDB is a NoSQL and thus a schema-free database. For the time being, the information is classified into five collections, namely:

1. flavourCollection
2. thingVisorCollection
3. userCollection
4. vSiloCollection
5. vThingCollection

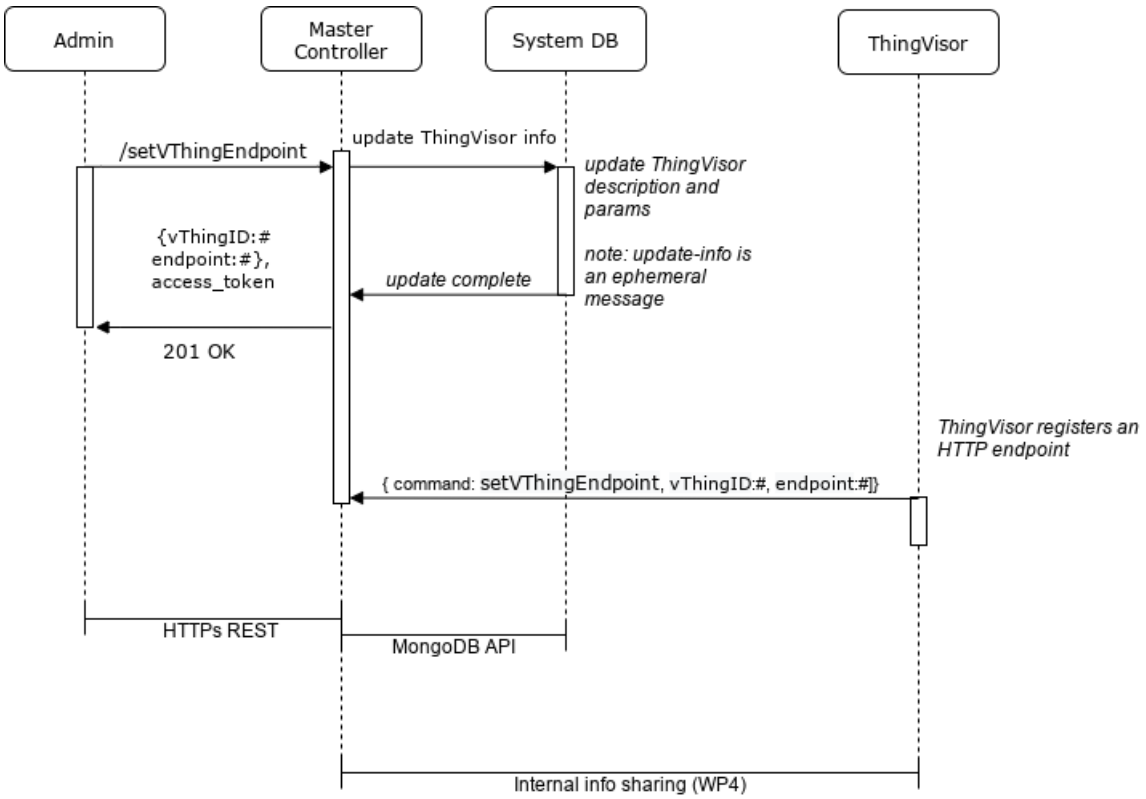


Figure 16: setVThingEndpoint ThingVisor procedure

Table 6 shows the general description of the collections in the case of a Kubernetes deployment of VirIoT, as well as a practical example for each one of them. In general, any collection can be added, removed and retrieved using the Command Line Interface. When new information must be stored, a new collection is added inside the database.

The `flavourCollection` stores information about the flavours, such as the list of image names the Master Controller employs to check and download from the Image Repository. Also, a list of yaml files is added to `yamlFiles`. In such a manner, the Master Controller is able to retrieve, from the database, the list of objects Kubernetes has to deploy when a new vSilo is added. The yamls are flavor-specific and can contain Deployments, StatefulSets and Services. Inside the `thingVisorCollection` information about the ThingVisors is stored, such as its instance (e.g. container) ID, IP address, a list of image names and all the vThings associated with it, as well as the IP address and port of the MQTT data and control broker, in case of internal information sharing based on MQTT (see D4.2). Also, additional information about the Kubernetes instance, such as the name of the deployment and service related to the ThingVisor are included, as well as details about multi-pod deployments. After the registration phase of the users is completed, the `userCollection` is used to save their credentials, namely, IDs, passwords and roles, along with their JSON Web Token, essential to the user in performing any action on the platform. The passwords are not in clear text, but an hashed version of them is stored. Lastly, the lastLogin field is updated whenever the user logs into the system. The `vSiloCollection` holds the information about the vSilos. In particular, other than the IP address, ports, instance (container) ID and image name, it contains the

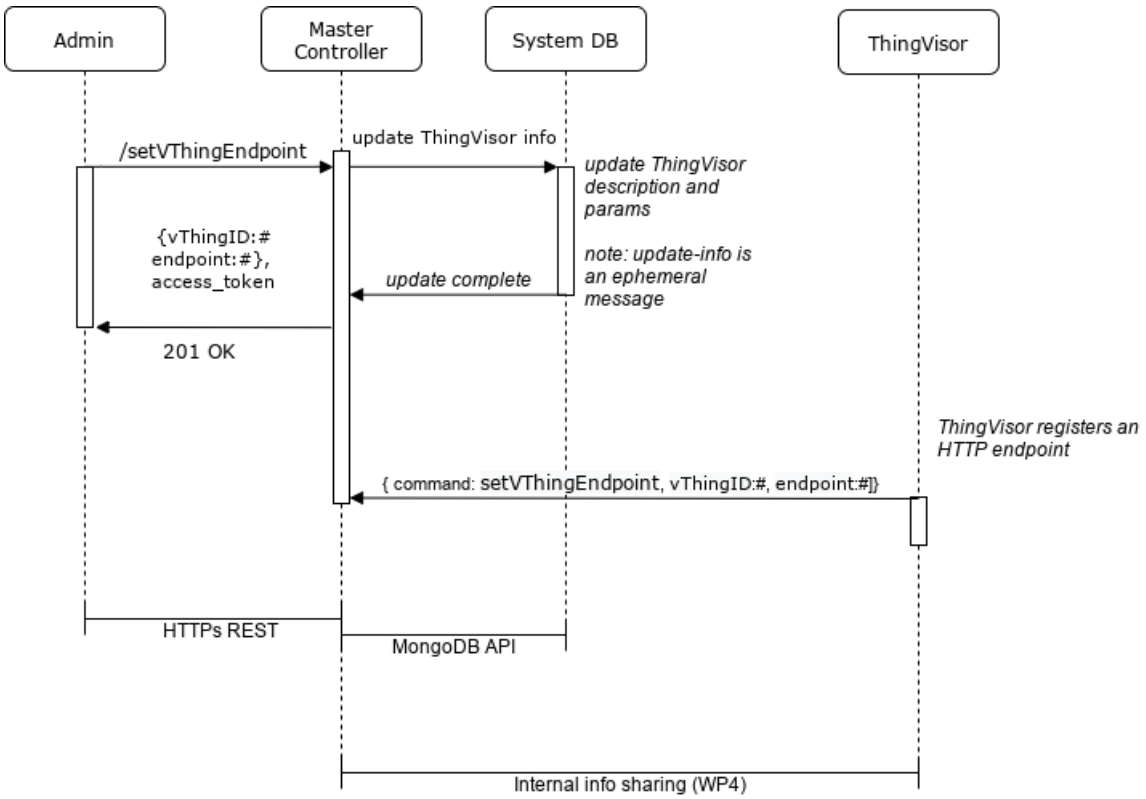


Figure 17: delVThingEndpoint ThingVisor procedure

user ID of the unique owner of the vSilo, the tenantID. The user is able to communicate with the vSilo throughout the IP address along with its port. Furthermore, in this collection, information about the Kubernetes deployment as well as the ThingVisor's Collection is present. Likewise, the collection that stores information about the vThings, the `vThingCollection`, contains the ID of the tenant and of the vSilo it is attached to, the vSiloID. It is a string formed by the tenant and vSilo ID, as it can be seen in the example in Table 6, "tenant1_Silo1".

```yaml
apiVersion: apps/v1 # for versions before 1.9.0 use apps/v1beta2
kind: Deployment
metadata:
  name: f4i-tv-relay
spec:
  selector:
    matchLabels:
      app: f4i-relay
  replicas: 1
  template:
    metadata:
      labels:
        app: f4i-relay
    spec:
      containers:
      - name: f4i-relay
        image: fed4iot/relay-tv
        ports:
        - containerPort: 8089
      - name: f4i-http-sidecar
        image: fed4iot/http-sidecar-tv
        ports:
        - containerPort: 5001
---
apiVersion: v1
kind: Service
metadata:
  name: f4i-tv-svc-relay
  labels:
    app: f4i-relay
spec:
  type: NodePort
  ports:
  - port: 8089
    targetPort: 8089
    nodePort:
    name: input
  - port: 80
    targetPort: 5001
    name: http
  selector:
    app: f4i-relay
```

Figure 18: Example of a Kubernetes YAML file of a Relay ThingVisor

Collection	JSON Description	Example

flavourC	'flavourID':flavour_id,
	'status':status,
	'flavourParams':
	flavour_parameter,
	'imageName':
	[
	image,
	],
	'flavourDescription':
	flavour_description,
	'creationTime':creation_time
	'yamlFiles':
	[
	{
	'apiVersion':apiversion,
	'kind':K8S_kind,
	'metadata':{},
	'spec':{}
	}
	]
		'flavourID':mobius-pub-sub-http-f,
		'status':ready,
		'flavourParams':Mobius,
		'imageName':
		[
		fed4iot/mobius2-pub-sub-actuator-flavour:
		lates,
		fed4iot/http-sidecar-flavour:latest
		]
		'flavourDescription':silo with a oneM2M
		Mobius broker,
		'creationTime':2021-02-21T18:33:26.626156
		'yamlFiles':
		[
		{
		'apiVersion':apps/v1,
		'kind':Deployment,
		'metadata':{},
		'spec':{}
		},
		{
		'apiVersion':v1,
		'kind':Service,
		'metadata':{},
		'spec':{}
		}
		]

thingVisorC	'thingVisorID':thing_visor_id,
	'status':status,
	'creationTime':creation_time,
	'tvDescription':tv_description,
	'containerID':container_id,
	'imageName':
	[
	image,
	],
	'ipAddress':ip_address,
	'debug_mode':debug,
	'vThings':
	[
	{
	'label':vt_label,
	'id':vt_id,
	'description':vt_description
	'endpoint':REST endpoint
	}
	],
	'params':parameters,
	'MQTTDataBroker':{
	'ip':data_broker_ip,
	'port':data_broker_port
	},
	'MQTTControlBroker':{
	'ip':control_broker_ip,
	'port':control_broker_port
	},
	'port':port,
	'IP':ip_address,
	'yamlFiles':
	[
	{
	'apiVersion':apiversion,
	'kind':K8S_kind,
	'metadata':{},
	'spec':{}
	}
	],
	'additionalDeploymentsNames':[],
	'additionalServicesNames':[],
	'deploymentName':deployment_name
	,
	'serviceName':service_name,
		'thingVisorID':relay-tv,
		'status':running,
		'creationTime':2021-02-21T18:36:00.651588,
		'tvDescription':Relay Thingvisor with Http,
		'containerID':f4i-tv-relay-relay-tv,
		'imageName':
		[
		fed4iot/relay-tv,
		f4i-http-sidecar
		],
		'ipAddress':f4i-tv-svc-relay-relay-tv.
		default.svc.cluster.local,
		'debug_mode':false,
		'vThings':
		[
		{
		'label':timestamp,
		'id':relay-tv/timestamp,
		'description':Pass Through vThing,
		'endpoint':http://192.168.122.1:8080
		},
		],
		'params':
		{
		'vThingName':timestamp,
		'vThingType':timestamp
		},
		'MQTTDataBroker':{
		'ip':vernemq-mqtt.default.svc.cluster.
		local,
		'port':1883
		},
		'MQTTControlBroker':{
		'ip':vernemq-mqtt.default.svc.cluster.
		local,
		'port':1883
		},
		'port':{
		'8089/tcp':32317,
		'80/tcp':30662,
		},
		'IP':160.80.82.44,
		'yamlFiles':
		[
		{
		'apiVersion':apps/v1,
		'kind':Deployment,
		'metadata':{},
		'spec':{}
		},
		{
		'apiVersion':v1,
		'kind':Service,
		'metadata':{},
		'spec':{}
		}
		],
		'additionalDeploymentsNames':[],
		'additionalServicesNames':[],
		'deploymentName':f4i-tv-relay-relay-tv,
		'serviceName':f4i-tv-svc-relay-relay-tv

userC	'userID':user_id, 'password':user_password, 'role':user_role, 'lastLogin':user_last_login, 'token':user_token	'userID':admin, 'password':pbkdf2:sha256:150000 $mWOSdeEB$09fa6, 'role':admin, 'lastLogin':2021-02-21T18:27:26.106571, 'token':eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9

vSiloC	'vSiloID':vsilo_id,
	'status':status,
	'creationTime':creation_time,
	'tenantID':tenant_id,
	'flavourParams':flavour_params,
	'containerName':container_name,
	'containerID':container_id,
	'ipAddress':vsilo_address,
	'port':vsilo_port,
	'vSiloName':vsilo_name,
	'flavourID':flavour_id,
	'MQTTControlBroker':{
	'ip':control_broker_ip,
	'port':control_broker_port
	},
	'MQTTDataBroker':{
	'ip':data_broker_ip,
	'port':data_broker_port
	},
	'additionalDeploymentsNames':[],
	'additionalServicesNames':[],
	'deploymentName':deployment_name
	,
	'serviceName':service_name
		'vSiloID':tenant1_Silo1,
		'status':running,
		'creationTime':2021-02-21T18:37:11.828105,
		'tenantID':tenant1,
		'flavourParams':'',
		'containerName':f4i-mobius-pub-sub-actuator-
		f-tenant1-silo1,
		'containerID':f4i-mobius-pub-sub-actuator-f-
		tenant1-silo1,
		'ipAddress':160.80.82.44,
		'port':
		{
		'7579/tcp':31906,
		'1883/tcp':30555,
		'80/tcp':31130
		},
		'vSiloName':Silo1,
		'flavourID':mobius-pub-sub-http-f
		'MQTTControlBroker':{
		'ip':vernemq-mqtt.default.svc.cluster.
		local,
		'port':1883
		},
		'MQTTDataBroker':{
		'ip':vernemq-mqtt.default.svc.cluster.
		local,
		'port':1883
		},
		'additionalDeploymentsNames':[],
		'additionalServicesNames':[],
		'deploymentName':f4i-mobius-pub-sub-actuator
		-f-tenant1-silo1,
		'serviceName':f4i-mobius-pub-sub-actuator-
		svc-tenant1-silo1

vThingC	'tenantID':tenant_id, 'vThingID':v_thing_id, 'creationTime':creation_time, 'vSiloID':v_silo_id	'tenantID':tenant1, 'vThingID':relay-tv/timestamp, 'creationTime':2021-02-21T18:39:16.396819, 'vSiloID':tenant1_Silo1

Table 6: Collections stored inside MongoDB

3.3 Developed ThingVisors

Table 7 summarizes the ThingVisors we have developed so far. The following sections give details about operation and implementation of a subset of them. However the GitHub website of the platform [2] includes a description of all of them. Many ThingVisors exploit the VirIoT functionality for virtualizing actuators (i.e. actuation-commands) we presented in D2.3.

ThingVisor Name	Short Description	Reference
HTTP Sidecar (section 3.3.1)	This is a container that can be added to any ThingVisor to enable the support of VirIoT HTTP services.	https://github.com/fed4iot/VirIoT/tree/master/Thingvisors/DockerThingVisor/ThingVisor_http_sidecar
Hello World	This is a dummy ThingVisor with the purpose of testing, that periodically sends context data every 5 seconds.	https://github.com/fed4iot/VirIoT/tree/master/Thingvisors/DockerThingVisor/ThingVisor_HelloWorld
Generic oneM2M (section 3.3.2)	This ThingVisor inject in VirIoT context data coming from an external oneM2M Mobius server. It fetches oneM2M content instances from a set of oneM2M containers and relay them into VirIoT as a single vThing, with many NGSI-LD entities (one per oneM2M container), and publishes latest container content instances as updates of NGSI-LD properties.	https://github.com/fed4iot/VirIoT/tree/master/Thingvisors/DockerThingVisor/ThingVisor_oneM2M_multiSub
OpenWeatherMap (section 3.3.9)	This ThingVisor is able to provide vThings that are virtual weather sensors (thermometer, barometer, etc.), for specified cities, by virtualizing information coming from the openweathermap.org service.	https://github.com/fed4iot/VirIoT/tree/master/Thingvisors/DockerThingVisor/ThingVisor_vWeather
Relay (section 3.3.10)	This ThingVisor creates a vThing whose data is JSON objects received from an external producer through a ThingVisor's HTTP `/notify` endpoint.	https://github.com/fed4iot/VirIoT/tree/master/Thingvisors/DockerThingVisor/ThingVisor_Relay
FIWARE-based Greedy (section 3.3.3)	This ThingVisor obtains all data entities coming from a remote Orion Context Broker and relays into the system as NGSI-LD entities.	https://github.com/fed4iot/VirIoT/tree/master/Thingvisors/DockerThingVisor/FIWARE-flavoured-ThingVisors. Specific files of this component in greedy-TV subfolder.
FIWARE-based Parking site (section 3.3.4)	This ThingVisor obtains data related to the domain of parking site availability for Smart Parking use case and relays into the system as NGSI-LD entities.	https://github.com/fed4iot/VirIoT/tree/master/Thingvisors/DockerThingVisor/FIWARE-flavoured-ThingVisors. Specific files of this component in Parkingsite-TV subfolder.
FIWARE-based Regulated Parking Zones (section 3.3.5)	This ThingVisor obtains data related to the domain of Regulated Parking Zones (RPZ) for Smart Parking use case and relays into the system as NGSI-LD entities.	https://github.com/fed4iot/VirIoT/tree/master/Thingvisors/DockerThingVisor/FIWARE-flavoured-ThingVisors. Specific files of this component in RPZ-TV subfolder.
FIWARE-based Aggregated Parking Value (section 3.3.6)	This ThingVisor obtains data related to the domain of parking site availability of Smart Parking use case and exposes an aggregated value (sum) of free parking spaces as NGSI-LD entities.	https://github.com/fed4iot/VirIoT/tree/master/Thingvisors/DockerThingVisor/FIWARE-flavoured-ThingVisors. Specific files of this component in AggrValue-TV subfolder.
FIWARE-based Actuator (section 3.3.7)	This ThingVisor obtains data coming from sources connected to a remote Orion Context Broker and supports actuation over entities that contain command attributes.	https://github.com/fed4iot/VirIoT/tree/master/Thingvisors/DockerThingVisor/FIWARE-flavoured-ThingVisors. Specific files of this component in Actuator-TV subfolder.
Wildlife Monitoring (section 3.3.8)	This ThingVisor virtualizes some sensors that are used for wildlife monitoring, such as Hygrometer, RainGauge, Illuminometer, and AnimalDetectors.	https://github.com/fed4iot/VirIoT/tree/master/Thingvisors/DockerThingVisor/ThingVisor_WildlifeMonitoring

Face Recognition (section 3.3.11)	The goal of this ThingVisor is to do face recognition by actuating a camera system, and to virtualize the camera system as a single face recognition device.	`https://github.com/fed4iot/VirIoT/tree/master/Thingvisors/DockerThingVisor/ThingVisor_FaceRecognition`
CameraBot (section 3.3.12)	The CameraBot ThingVisor is a ThingVisor able to interact with a robot in order to control its behaviour in terms of virtualization of its on-board video camera, exposing it as a set of configurable "static cameras".	`https://github.com/fed4iot/VirIoT/tree/master/Thingvisors/DockerThingVisor/ThingVisor_CameraBot`
Philips Hue (section 3.3.13)	The goal of this ThingVisor is to do face recognition by actuating a camera system, and to virtualize the camera system as a single face recognition device.	`https://github.com/fed4iot/VirIoT/tree/master/Thingvisors/DockerThingVisor/ThingVisor_Philips_Hue`. Specific files for this component are in the Test subfolder. Also, in the Extra top-level folder, a copy of the Hue Emulator is found.
LoRaWAN (section 3.3.14)	This ThingVisor helps to bind vThings with real devices living in a LoRaWAN network managed by a Chirpstack server.	`https://github.com/fed4iot/VirIoT/tree/master/Thingvisors/DockerThingVisor/ThingVisor_LoRaWAN`
FogFlow (section 3.3.15)	This ThingVisor is the glue between the platform and the ThingVisor Factory based on FogFlow	`https://github.com/fed4iot/VirIoT/tree/master/Thingvisors/DockerThingVisor/ThingVisor_FogFlow`

Table 7: ThingVisors Portfolio

3.3.1 HTTP ThingVisor Sidecar

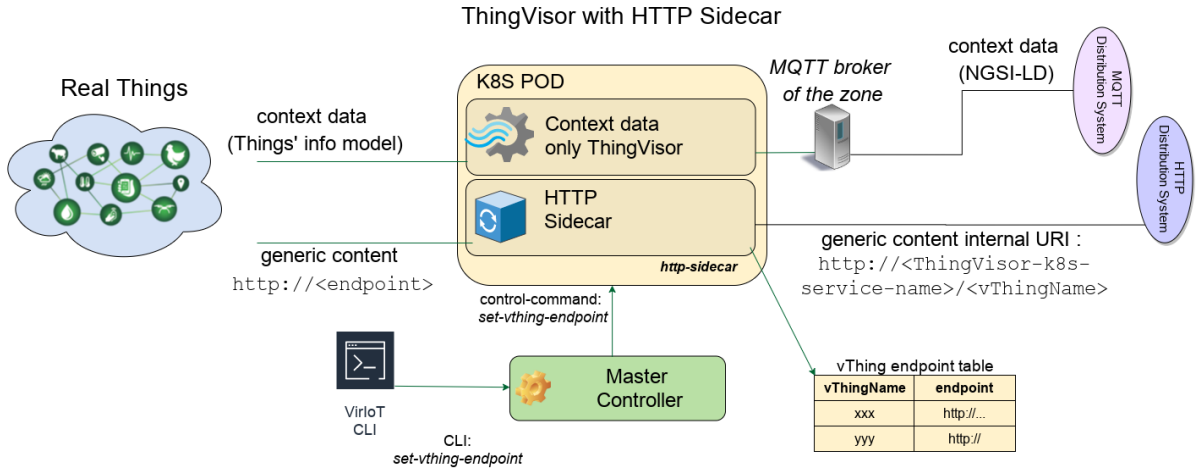


Figure 19: HTTP sidecar for ThingVisors

The HTTP ThingVisor Sidecar is not actually a stand-alone ThingVisor but is a container that can be added to any ThingVisor without native HTTP support to add it to them. Currently, any developed ThingVisor uses this sidecar to offer HTTP services. Consequently, we only present their context data related functionality.

Figure 19 shows the architecture of a ThingVisor that includes the HTTP sidecar. The HTTP sidecar is a process that receives HTTP GET/POST on the internal URI of a vThing, `http://<ThingVisor-K8S-service-name>/<vThingName>`, and forwards this GET/POST to an external HTTP endpoint by using an internal table whose entries are configured by using the setVthingEndpoint procedure.

3.3.2 Generic oneM2M ThingVisor

This ThingVisor is able to connect to data sources that follow the oneM2M standard, and it obtains data pieces coming from a set of specified oneM2M Containers that are hosted on one or more remote oneM2M platforms.

The following box shows a typical set of parameters that can be used to configure the ThingVisor during the "Add ThingVisor" operation (please see Section 3.1.13 to understand how the "Add ThingVisor" operation is performed).

Typical **oneM2M ThingVisor** Parameters

```
{'CSEurl':'https://fed4iot.eglobalmark.com','origin':'Superman', 'poaPort
    ':'8089','cntArns':['Abbas123456/humidity/value','Abbas123456/
    batteryLevel/value'],'poaIP':'52.166.X.X','vThingName':'EGM-
    Abbas123456-humidity','vThingDescription':'OneM2M humidity data from
    EGM Abbass sensor'}
```

The above configuration tells the ThingVisor to connect to a oneM2M system located at our EGM partner's infrastructure, and to subscribe to the Abbas123456 sensor's humidity readings. This humidity sensor is going to be virtualized inside our platform as a vThing whose ID is `EGM-Abbas123456-humidity`, and it comprises the actual value of humidity and the battery level of the sensor itself. The `poaIP` is the public IP address of our platform, so that the remote oneM2M system (a Mobius broker is this specific EGM deployment) knows how to notify the ThingVisor of new data.

Indeed, this ThingVisor is based on a oneM2M subscribe mode of operation, and at startup time it registers an own notification end-point into the remote oneM2M platforms, so as to be notified by the remote oneM2M systems whenever new data is produced for the specified Containers. During the initialization phase, the ThingVisor also requests the latest ContentInstances available inside all specified oneM2M Containers, so as to be ready to reply to possible getContextRequest commands coming from vSilos. Eventually, it announces the newly created vThing to the downstream components of the VirIoT system, specifically to the Master Controller (see Section 3.2.4 and Figure 11).

Upon arrival of a fresh piece of data, the ThingVisor carries out a translation from oneM2M to NGSI-LD and re-publishes the data internally, within VirIoT, making it available for downstream vSilos. The high-level operation is shown in Figure 20.

Thus, this ThingVisor is central to achieve seamless interoperability with oneM2M systems, in the forward direction (from oneM2M to other standards/platforms). Importantly, the mapping from oneM2M data to NGSI-LD data (i.e. to our neutral internal format) is **not automatic as the reverse one is** (i.e. the one from our neutral format to oneM2M, which is carried out along the mapping procedures we have specified in D2.2). The ThingVisor developer (or the ThingVisor logic) must know (or query) the underlying information model and resource structure of the oneM2M system, and decide what Containers are to be grouped together under the same vThing. The important thing to remark here is that this ThingVisor is capable of exploiting the (reverse of the) "one vThing" to "many Containers" mapping guideline that we have specified in D2.2.

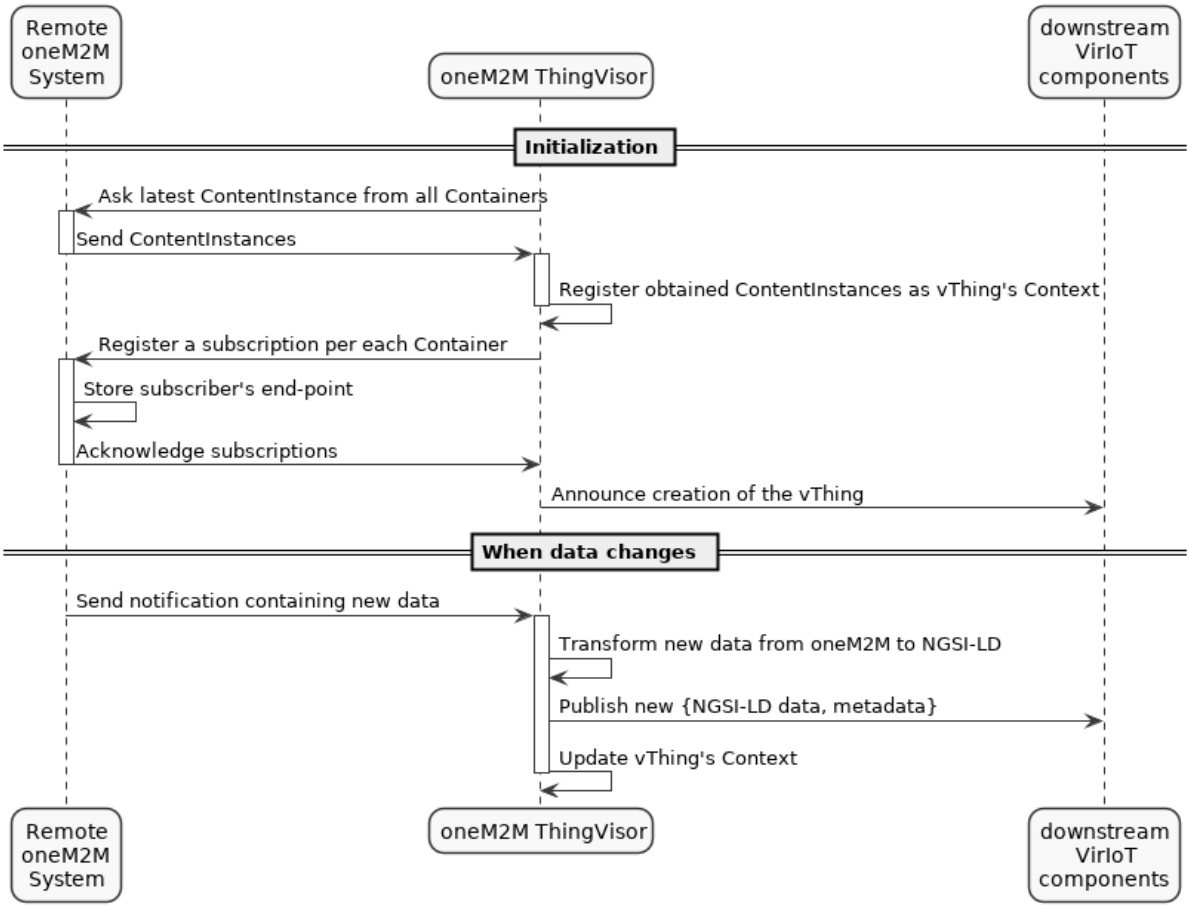


Figure 20: oneM2M ThingVisor

It is then possible to configure (as seen above) the ThingVisor with a list of oneM2M Containers that are to be grouped under the same vThing within VirIoT.

3.3.3 FIWARE-based Greedy ThingVisor

This ThingVisor is able to connect to data sources that follow the NGSIv2 standard, and it obtains data entities coming from a remote Orion Context Broker.

The following box shows a typical set of parameters that can be used to configure the ThingVisor during the "Add ThingVisor" operation (please see Section 3.1.13 to see how the "Add ThingVisor" operation is performed).

Typical **Greedy ThingVisor** Parameters

```
{'ocb_service':['trafico','aparcamiento','pluviometria','tranvia','
    autobuses','bicis','lecturas','gps','suministro'], 'ocb_ip':'fiware-
    dev.inf.um.es', 'ocb_port':'1026'}
```

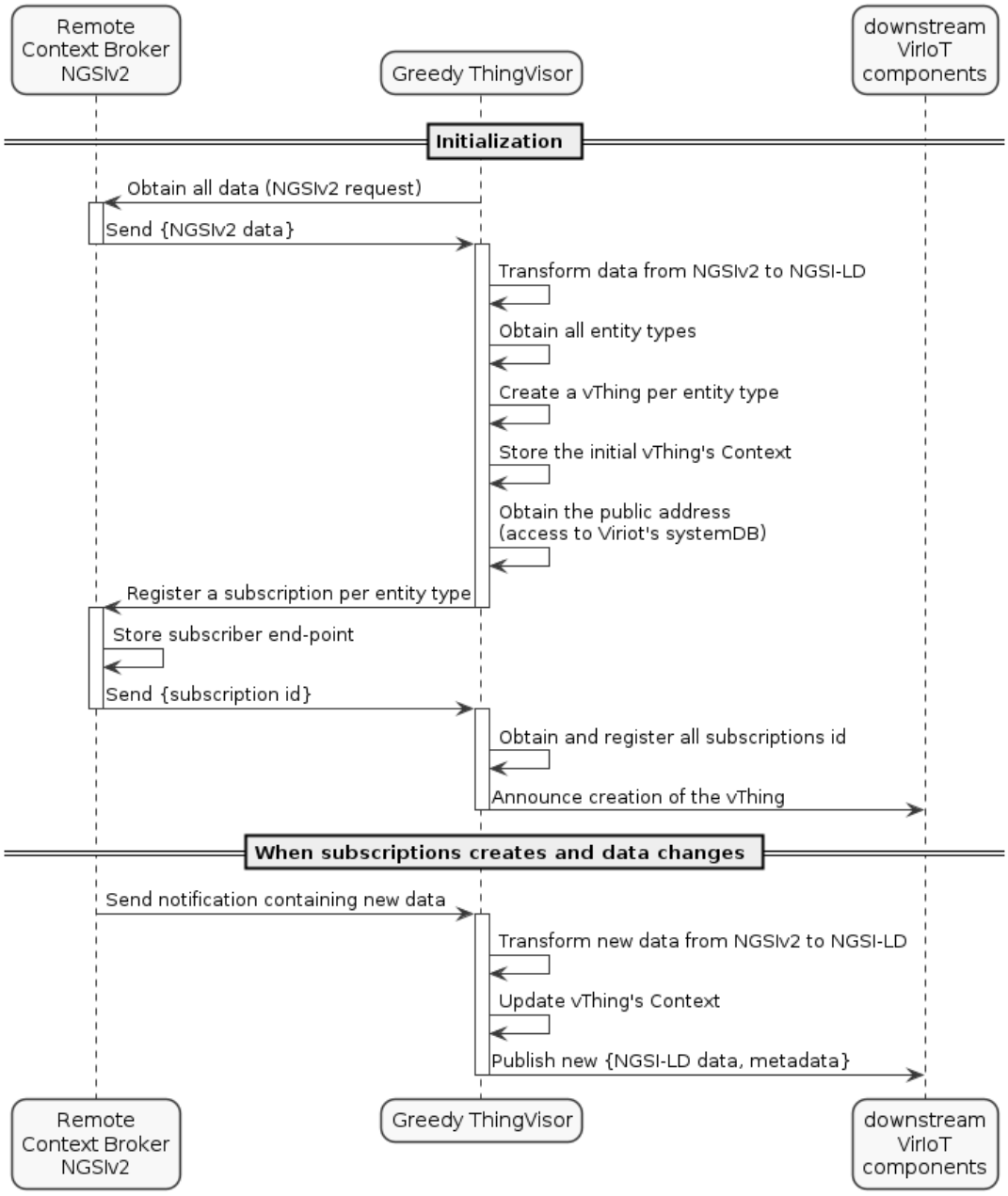


Figure 21: Generic NGSIv2 Greedy ThingVisor

The above configuration configures the ThingVisor to connect to a remote FIWARE-based platform which exposes an NGSIv2 API, and more specifically to an Orion Context Broker GE (Generic Enablers in FIWARE terminology build an ecosystem of applications, services and data), to obtain all entities of each `ocb_service` defined. Orion Context Broker GE offers multiple isolated data reporting environments for different purposes with a single broker, in this sense, `ocb_service` allows you to define an array that this ThingVisor will use to access all information of these environments. This information is going to be virtualized inside our platform by using different vThings, specifically one vThing per NGSIv2 entity type the remote FIWARE platform contains.

Furthermore, this ThingVisor uses publication and subscription mechanisms to be updated every time the information changes in the selected remote Context Broker. The ThingVisor accesses to the System DB of Viriot platform to obtain its public IP address and port. This one is required to receive the notifications from the Context Broker. Once this ThingVisor get it, it sends a subscription request per entity type, to the remote Context Broker.

To close the initialization phase, ThingVisor sends `createVThing` messages about the new vThings to the downstream components of the VirIoT system, specifically to the Master Controller (see Section 3.2.4 and Figure 11).

When the ThingVisor receives the remote Context Broker notifications, it transforms the data from NGSIv2 to neutral NGSI-LD format, as mentioned in Deliverable 4.2, and re-publishes the data internally, within VirIoT, making it available for downstream vSilos. The high-level operation is shown in Figure 21.

3.3.4 FIWARE-based Parking site ThingVisor

This section includes a ThingVisor that is relevant for our Smart Parking use case, in this case this, the ThingVisor recovers data related to the domain of parking site availability (sensors and policies).

This component is able to connect to data sources that follow the NGSIv2 standard, and it obtains data entities coming from a remote Orion Context Broker (FIWARE platform).

The following box shows a typical set of parameters that can be used to configure the ThingVisor during the "Add ThingVisor" operation (please see Section 3.1.13 to understand how the "Add ThingVisor" operation is performed). In this case, parameters of this box configure the ThingVisor to access at the Orion Context Broker of the Murcia FIWARE platform.

Typical **Parking site ThingVisor** Parameters

```
{'ocb_ip':'fiware-dev.inf.um.es', 'ocb_port':'1026'}
```

Figure 22 shows how this component obtains data that is relevant to our Smart Parking use case from a particular Orion Context Broker that supports NGSIv2 API, and it exposes the corresponding payloads to other VirIoT components, by transforming them to our neutral NGSI-LD format.

The internal functionality of this ThingVisor is the same as mentioned above for the Greedy one, but this specific ThingVisor focuses only towards those data pieces that are relevant information for the Smart Parking use case.

3.3.5 FIWARE-based Regulated Parking Zones ThingVisor

This section includes another ThingVisor that is relevant for our Smart Parking use case, in this case this, the ThingVisor recovers data related to the domain of Regulated Parking Zones (parkingmeters, city sectors and policies).

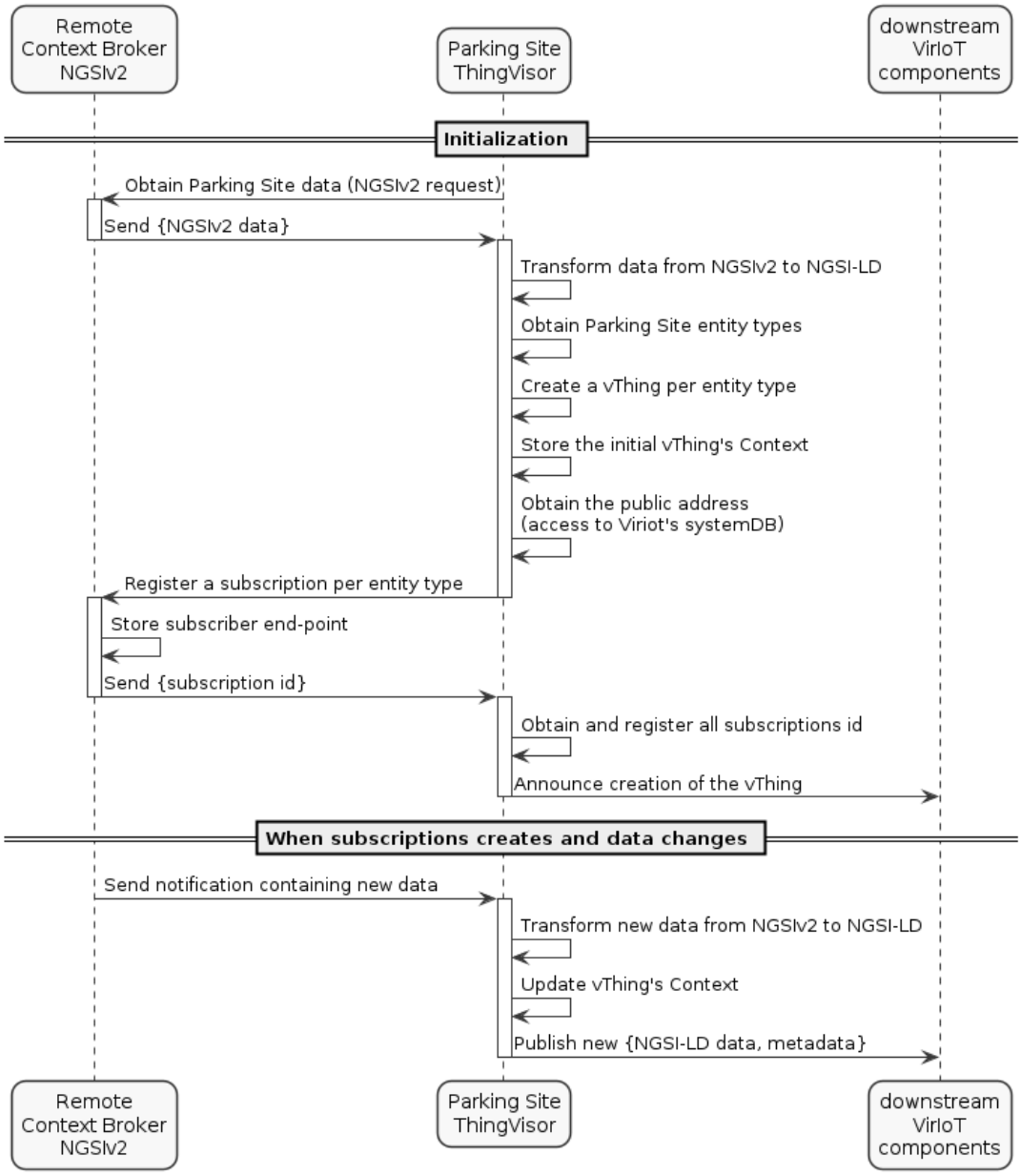


Figure 22: Parking site ThingVisor

This component is able to connect to data sources that follow the NGSIv2 standard, and it obtains data entities coming from a remote Orion Context Broker (FIWARE platform).

The following box shows a typical set of parameters that can be used to configure the ThingVisor during the "Add ThingVisor" operation (please see Section 3.1.13 to understand how the "Add ThingVisor" operation is performed). In this case, parameters of this box configure the ThingVisor to access at the Orion Context Broker of the Murcia FIWARE platform.

Typical **Regulated Parking Zones ThingVisor** Parameters

```
{'ocb_ip':'fiware-dev.inf.um.es', 'ocb_port':'1026'}
```

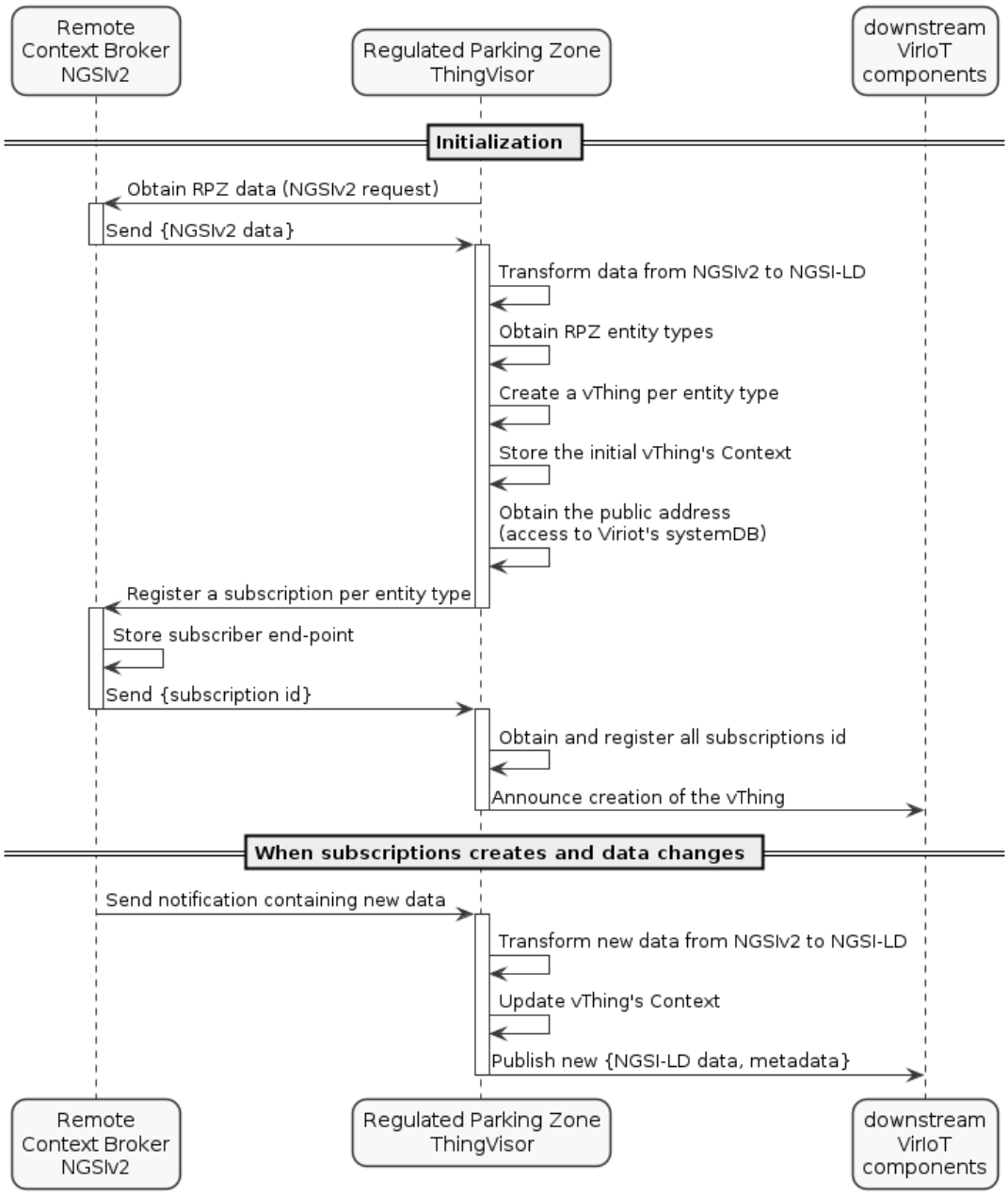


Figure 23: Regulated Parking Zones ThingVisor

Figure 23 shows how this component obtains data that is relevant to our Smart Parking use case from a particular Orion Context Broker that supports NGSIv2 API, and it exposes the corresponding payloads to other VirIoT components, by transforming them to our neutral NGSI-LD format.

The internal functionality of this ThingVisor is the same as mentioned above for the Greedy one, but this specific ThingVisor focuses only towards those data pieces that are relevant information for the Smart Parking use case.

3.3.6 FIWARE-based Aggregated Parking Value ThingVisor

This component is able to connect to data sources that follow the NGSIv2 standard, and it obtains data entities coming from a remote Orion Context Broker (FIWARE platform).

The following box shows a typical set of parameters that can be used to configure the ThingVisor during the "Add ThingVisor" operation (please see Section 3.1.13 to understand how the "Add ThingVisor" operation is performed).In this case, the parameters of this box configure the ThingVisor to access at the Orion Context Broker of the Murcia FIWARE platform.

Typical **Aggregated Value ThingVisor** Parameters

```
{'ocb_ip':'fiware-dev.inf.um.es', 'ocb_port':'1026'}
```

Figure 24 shows how this component obtains data information relevant for the Smart Parking use case from a particular Orion Context Broker, processes and exposes an aggregated value in neutral NGSI-LD format. At this stage, the aggregated value is simply the sum of free parking spaces. In order to accomplish the task, this ThingVisor receives data whenever there are changes, using publication and subscription mechanisms, and recalculates the aggregated value. Periodically, it verifies whether or not the aggregated value has changed since it was last published to other VirIoT components; only when this occurs, then ThingVisor re-publishes the data. The ThingVisor only needs one virtual thing to carry out its task.

3.3.7 FIWARE-based Actuator ThingVisor

This component is able to connect to data sources that follow the NGSIv2 standard. It obtains data entities coming from a remote Orion Context Broker (FIWARE platform) and supports actuation over entities that contain command attributes.

The following box shows a typical set of parameters that can be used to configure the ThingVisor during the "Add ThingVisor" operation (please see Section 3.1.13 to understand how the "Add ThingVisor" operation is performed). In this case, the parameters of this box configure the ThingVisor to access at a generic remote Orion Context Broker and optionally can specify the fiware-service and fiware-servicepath where recover entities information inside the broker.

Typical **Actuator ThingVisor** Parameters

```
{'ocb_ip':'<RemoteCB-Host>', 'ocb_port':'<RemoteCB-Post>', 'ocb_service
    ':['service1', 'service1',...], 'ocb_servicePath':['servicepath1', '
    servicepath2',...]}
```

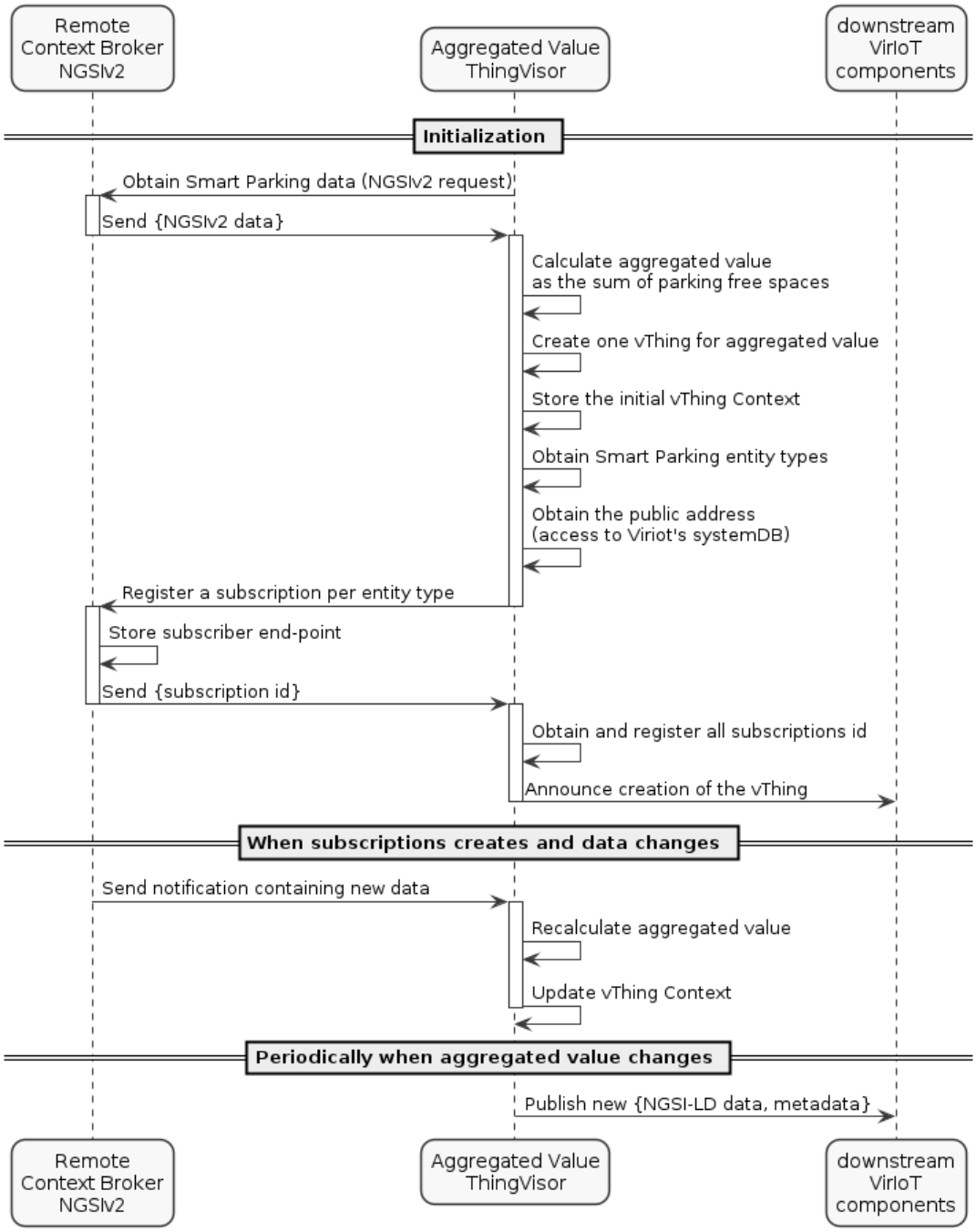


Figure 24: Aggregated Parking Value ThingVisor

Figure 25 shows how this component obtains data information of IoT Devices represented at a remote Orion Context Broker that supports NGSIv2 API, and they expose the corresponding payloads to other VirIoT components, by transforming them to our neutral NGSI-LD format.

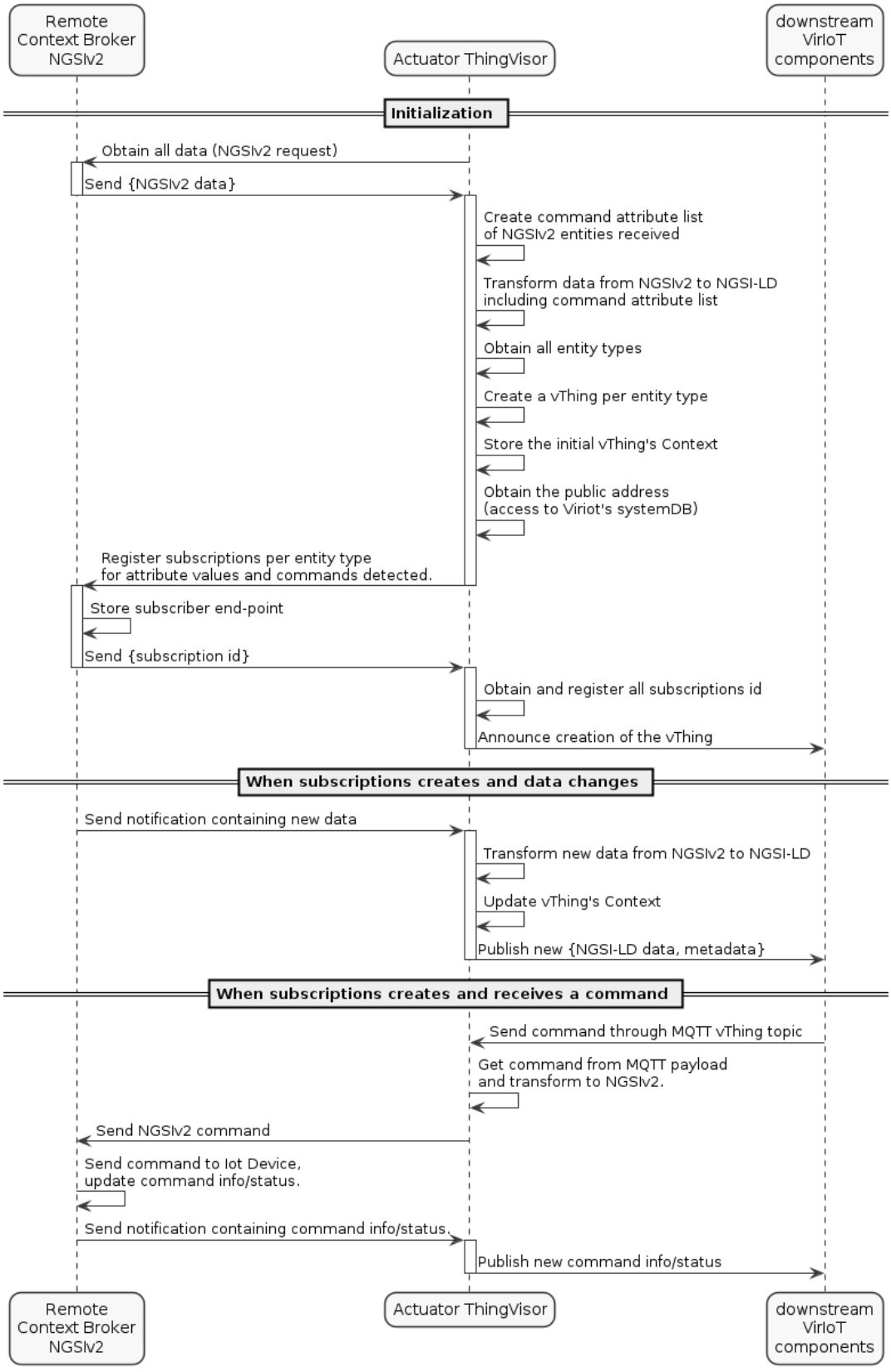


Figure 25: Actuator ThingVisor

Generically, the internal functionality of this ThingVisor is the same as mentioned above for the Greedy one, but additionally, this ThingVisor has some steps to supporting actuation-command functionality. In the initialization process, this component discovers actuation-commands included at the remote Orion Context Broker entities, as mentioned above, these entities represent IoT Devices.

These entities include their corresponding actuation-commands list in NGSI-LD context of the ThingVisor which sends additional subscriptions to the remote Context Broker by detected actuation-command.

In this sense when an actuation-command arrives at the ThingVisor, it transforms MQTT actuation-command payload to NGSIv2 and sends the actuation-command to the corresponding entity of remote Context Broker and, finally when info/status values change the notification arrives at the ThingVisor.

3.3.8 Wildlife Monitoring ThingVisor

This ThingVisor virtualizes some sensors that are used for wildlife monitoring, such as Hygrometer, RainGauge, Illuminometer, and AnimalDetectors. This component is able to connect to data sources that follow the NGSIv2 standard, and it obtains data entities coming from a remote Orion Context Broker (FIWARE platform).

Figure 26 shows how this component obtains data information relevant for the Wildlife Monitoring use case from a particular Orion Context Broker that supports NGSIv2 API, and it exposes the corresponding payloads to other VirIoT components, by transforming them to our neutral NGSI-LD format.

3.3.9 OpenWeatherMap ThingVisor

This ThingVisor is able to provide virtual weather sensors (thermometer, barometer, etc.), for specified cities, by virtualizing information coming from the openweathermap.org service.

We have registered a Fed4IoT account on the openweathermap.org open API system, which allows our ThingVisors to periodically poll selected information from the openweather service, based on a custom configuration that is fed to the ThingVisor at startup time. Consequently, the OpenWeatherMap ThingVisor is able to create a set of vThings that act as virtual sensors for each measurable property (i.e. humidity, temperature, current atmospheric pressure) for each of the locations that have been specified at ThingVisor creation time.

The following box shows a typical set of parameters that can be used to configure the ThingVisor during the "Add ThingVisor" operation (please see Section 3.1.13 to understand how the "Add ThingVisor" operation is performed).

Typical **OpenWeatherMap ThingVisor** Parameters

```
{'cities':['Rome', 'Tokyo','Murcia','Grasse','Heidelberg'], 'rate':60}
```

The above configuration tells the ThingVisor to poll the openweather APIs every 60s,

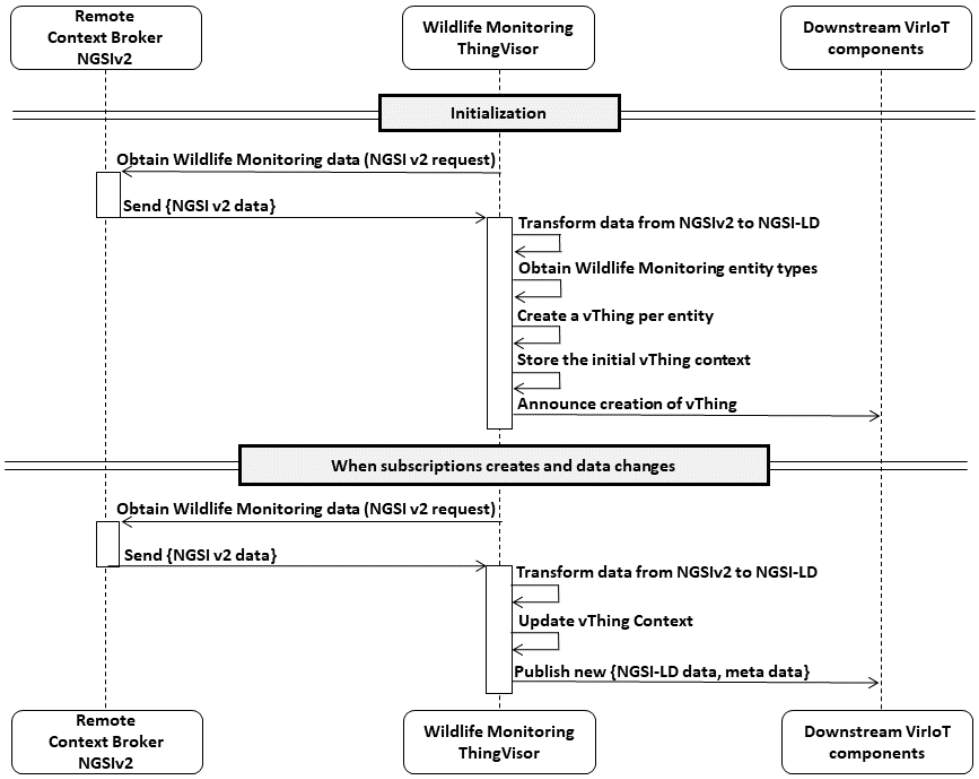


Figure 26: Wildlife Monitoring ThingVisor

and it tells that the ThingVisor is to create a set of virtual sensors for all the specified cities and all the available measurable properties.

Figure 27 shows the flow of operations of the OpenWeatherMap ThingVisor.

3.3.10 Relay ThingVisor

The Relay-ThingVisor implements a simple dummy virtual sensor, which relays back into the system context data received from an external producer. Specifically, this ThingVisor creates a vThing whose data is JSON objects received from an external producer through the ThingVisor's HTTP `/notify` endpoint, that is on listen, internbally, at port 8089. It can be used as a test tool, which is able to continuously intake generic data, produced in the form of JSON objects. It injects them in the platform as NGSI-LD context information generated by its vThing.

3.3.10.1 Thingvisor parameters

The vThingName and the vThingType can be customized. The Relay TV accepts the following parameters:

- **vThingName**: vThing's name that identifies the virtual sensor.
- **vThingType**: The semantic type of the data produced by the vThing.

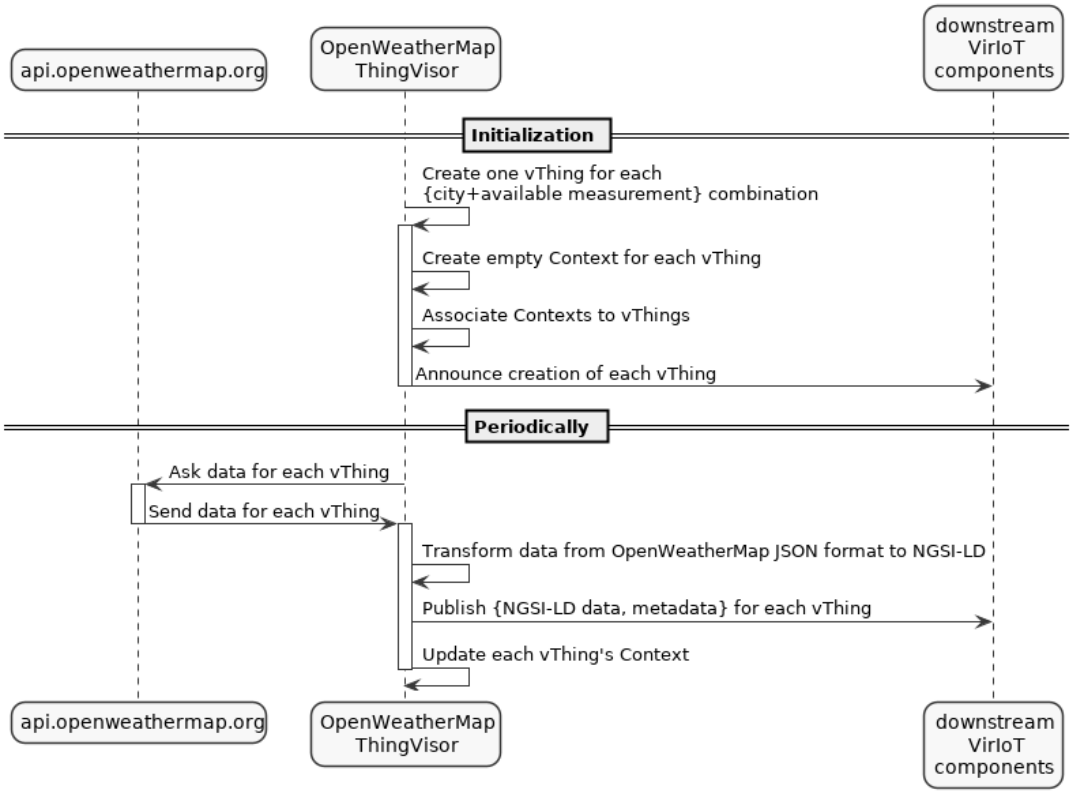


Figure 27: OpenWeatherMap ThingVisor

3.3.10.2 Data injection

The `curl`, among others, can be used to push a JSON object as in the following example:

Example command to inject data via Relay ThingVisor

```
curl -d '{"timestamp": 1594982023328, "sqn": 66941}' -H "Content-Type:
    application/json" -X POST http://<ThingVisorIP:ThingVisorPort>/notify
```

The NGSI-LD entity published by the vThing is as follows:

```
{
    "id": "urn:ngsi-ld:<ThingVisorName>:<vThingName>",
    "type": "<vThingType>",
    "msg": {
        "type": "Property",
        "value": <received JSON object>
    }
}
```

Thus it is to be noted that the virtual sensor produces NGSI-LD entities that have only one property, called `msg`, which contains the injected JSON object.

3.3.11 Face Recognition ThingVisor

The goal of this ThingVisor is to do face recognition by actuating a camera system, and to virtualize the camera system as a single face recognition device. To do this, it must have a set of target pictures of various persons (base pictures) and their respective names. These images must be sent to the camera system's face recognition process.

The camera system does face recognition and sends the results back to the ThingVisor through a REST interface. Results of the recognition process and constant updates about its current status are available to VirIoT through the MQTT and HTTP data distribution systems of the platform.

The Thingvisor creates one single Virtual Thing, called detector. This vThing is an actuator, so the user can send actuation-commands to the detector vThing through the vSilo, and receive actuation-command results. Specifically, as soon as the detector is added to a vSilo, the vSilo's users can send actuation-commands to the TV through the vSilo's actuation interface, and they can send the target base pictures to the TV through vSilo's sidecar HTTP interface.

Internally, the TV needs to talk to the camera system, in order to send information for face recognition and receive recognition results.

Figure 28 is a diagram showing the communications between vSilo, ThingVisor, and the camera system in the basic operations that involve sending a picture of a person for face recognition and receiving face recognition results.

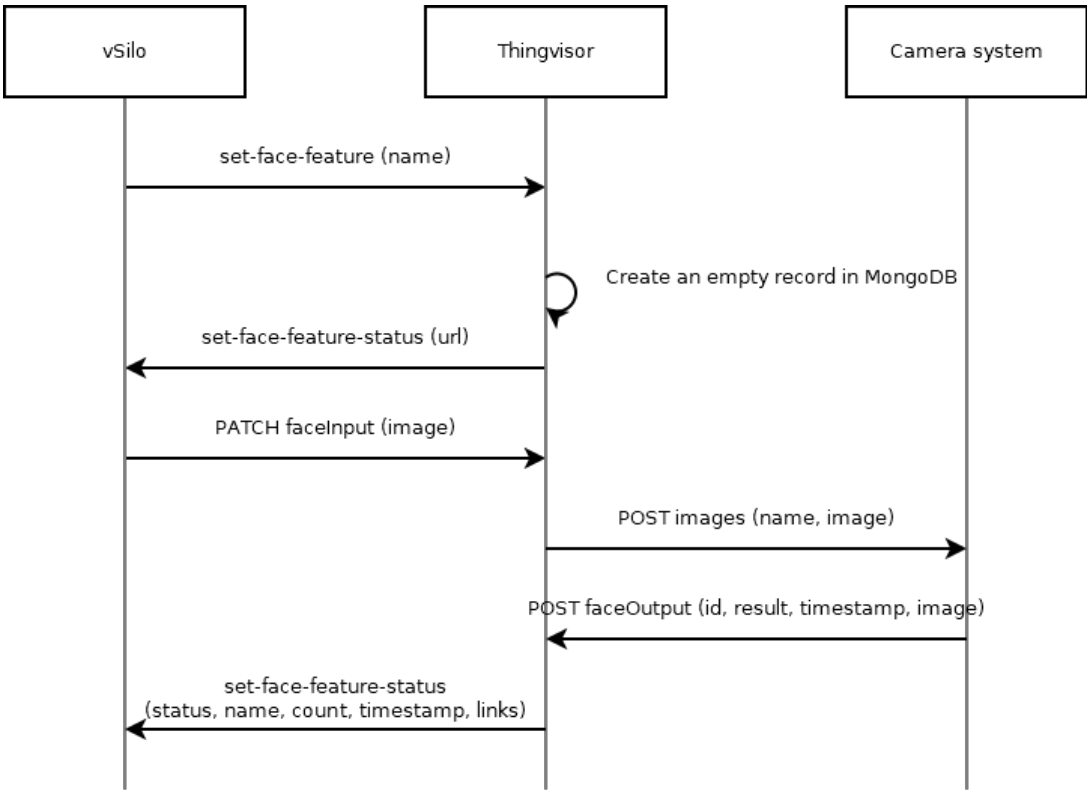


Figure 28: Face Recognition ThingVisor

1. The user wants to add a new picture for a person with a given name. So, she triggers the detector, which sends the set-face-feature actuation-command to the TV.

2. The TV creates a new record in its MongoDB database with a random id, then it sends a response to the user, that includes the URL where to patch the picture, via the set-face-feature-status associated to the set-face-feature actuation-command.

3. When the user patches the image to TV, the TV sends image and name to the camera system. The camera system processes the image, and stores face encoding.

4. When the camera system detects a status change (a person that was not in the image, now there is, and vice versa), it posts on TV the current image and the new status.

5. Then, the TV sends back a set-face-feature-status each time a status change (target face detected in the camera view, target face exits the camera view) occurs.

3.3.11.1 Sending actuation-commands to TV

When the detector vThing is added to a vSilo, an NGSI-LD entity is created and sent to the broker. Let's look at the NGSI-LD entity that represents the detector:

```
{
    "id": "urn:ngsi-ld:facerec:detector",
    "type": "FaceDetector",
    "commands": {"type": "Property", "value": ["start","stop",
                "set-face-feature","delete-by-name"]},
    "@context": [
        "https://uri.etsi.org/ngsi-ld/v1/ngsi-ld-core-context.jsonld"
    ]
}
```

For each actuation-command, the Silo controller creates the associated property. In order to send an actuation-command, modifying the associated property via the vSilo's broker is required.

Command values are JSON objects with the following properties:

- **cmd-value**: parameters of the command (mandatory)
- **cmd-qos**: required QoS (optional, default=0)
- **cmd-id**: unique command ID (mandatory)
- **cmd-nuri**: vSilo's unique notification URI, where to send NGSI-LD status/results

Whenever a user modifies a property representing an actuation-command, this is notified to the Silo controller, that will send it to the TV. The TV will execute the actuation-command, by doing whatever internal communication to the camera system is needed.

Here is a list of all available actuation-commands:

- **start**

  Send the start actuation-command to the camera system. QoS must be 0 or 1.

- **stop**

  Send the stop actuation-command to the camera system. QoS must be 0 or 1.

- **set-face-feature**

  Request for sending a new picture about a specific person. `cmd-value` must be an object containing the name field. QoS must be 2.

- **delete-by-name**

  Allows to delete all the pictures associated to a given name. `cmd-value` must be an object containing the name field. QoS must be 0 or 1.

The most important `cmd-status` is the status that is sent back in response to a set-face-feature actuation-command, i.e. set-face-feature-status; it a JSON object composed of the following fields:

- **status** (0 = person is present in the current image, 1 = person is not present)
- **name** (name of the person related to status change)
- **count** (number of base pictures related to the person)
- **timestamp**
- **link_to_base_image** (url where the user can GET via HTTP the target base image. Relative to the base prefix of the vThing's endpoint)
- **link_to_current_image** (url where the user can GET via HTTP the detected image. Relative to the base prefix of the vThing's endpoint)

An example cmd-status is as follows:

```
"cmd-status": {
  "status": 1,
  "name": "Andrea",
  "count": 3,
  "timestamp": <timestamp>,
  "link_to_base_image": "/faceInput/<uuid>/image",
  "link_to_current_image": "/faceOutput/<uuid>/image"
}
```

3.3.11.2 External HTTP endpoints for streaming pictures

Right after the ThingVisor starts up, and it initializes its detector vThing, it talks to the Master Controller in order to automatically set up the HTTP endpoint to be used to reach the vThing's picture streams. The following HTTP base prefix is automatically configured, so as to allow vSilo's users to obtain access to all the external HTTP picture stream endpoints of the detector, through the vSilo's sidecar HTTP interface:

`http://vsilo_ip:port/vstream/facerec/detector/<endpoint>`

Thus, the detector vThing is now able to expose the following REST endpoints to the vSilo users, via the platform's HTTP data distribution system:

- **[PATCH] /faceInput/<id>**

  The faceInput endpoint is for sending pictures of the known persons. Users have to send a form-data type body with a field named data, that will contain the picture of the person.

  Note: there is a single record for every picture. If Users patch two times with the same record id, the picture will be overwritten.

- **[POST] /genericFaceInput/<name>**

  This endpoint has the same purpose of the faceInput endpoint, but it works with person names, instead of image ids. Every time users post by name, a new record is generated. No image overwrite is possible.

- **[GET] /vstream/facerec/detector/faceInput/<id>/image**

  This endpoint allows the user to retrieve input images by id.

- **[GET] /vstream/facerec/detector/faceOutput/<id>/image**

  This endpoint allows the user to retrieve output images by id.

3.3.11.3 Internal endpoints to the camera system

When the camera system is started, we can pass IP and port of the ThingVisor as parameters. In this way, the camera system can post to the ThingVisor. At the beginning, the camera system presents itself to the TV, by sending its IP and port. However, this behaviour can be overriden by passing camera system IP and port directly to the TV as parameters when it's being created.

The ThingVisor exposes the following REST endpoints to the camera system:

- **[PATCH] http://thingvisor_ip:port/camera/0**

  The camera system can patch this endpoint to send to ThingVisors its IP and port.

- **[POST] http://thingvisor_ip:port/faceOutput**

  The faceOutput endpoint is used by the camera system to post to the TV status changes about the people, and the related images. These changes are then forwarded to the vSilo by the detector.

3.3.11.4 Thingvisor parameters

The FaceRecognition TV accepts the following parameters:

- **camera_ip**: The camera system IP.
- **camera_port**: The port to which the camera system running script is listening.
- **controller_url**: This is the URL of the master controller. It is used to automatically configure HTTP endpoints.
- **admin_id**: Username for logging in to the master controller.
- **admin_psw**: Password for logging in.

3.3.11.5 Implementation of the camera system

The software that runs on the camera system is a script written in Python.

The script initializes the main camera video stream with the following parameters:

- **Frame size**: 600 x 600 px
- **FPS**: 1

So, 1 time per second a callback is called.

The code is rather simple, because we are using the face-recognition 1.3.0 library (available on PyPI website). First of all, we use the face_recognition library to find the faces in the image. For every face, the respective face encoding is computed and saved in memory.

Then, we use again face_recognition to find matches between the encoding of every face found and the encodings of the all the base pictures. If one or more matches are found, the face with nearer distance is chosen.

The camera system keeps track of the face recognition results: for every base picture, the status will be 1 if the person is present in the current image, 0 if not.

While the TV stores people information in dictionaries, in the camera system script we used arrays.

This approach is due to the face-recognition library, that works with arrays.

The arrays are:

- **known_faces_encodings**
- **known_faces_names**
- **known_faces_ids**

Results are instead stored in the results dict, and referenced by face ids.

3.3.11.6 Supported camera systems

The Python script can be adapted to various camera systems. Currently it was tested on:

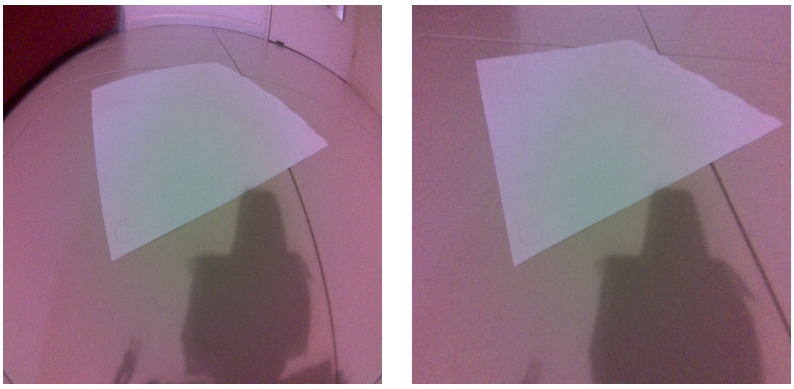
- JetBot from NVIDIA
- A notebook with a standard webcam

Python is supported on multiple platforms, so the script is basically cross-platform. The only thing that changes between these systems is the camera. Therefore, some additional code needs to be written for some systems, in order to support different camera handling.

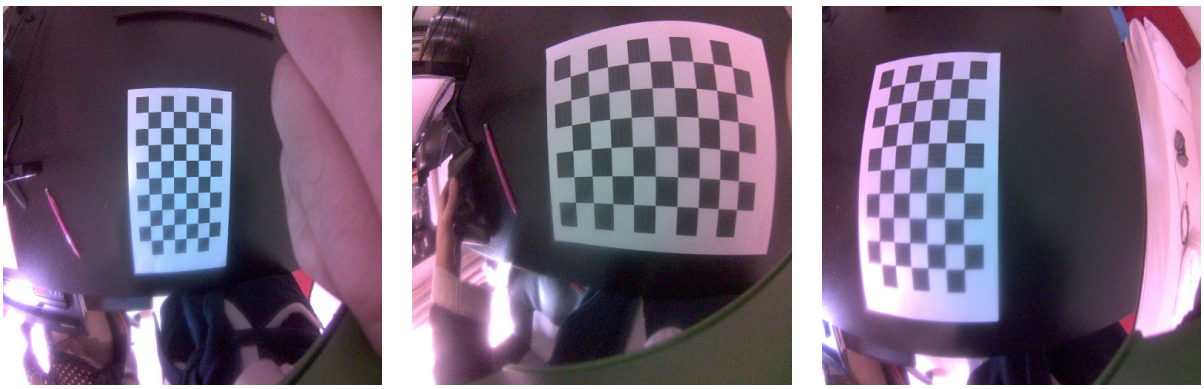
3.3.11.7 Camera calibration

The left image is a picture taken from the main camera, about a perfectly square piece of paper. As we can see, the four sides of the paper are curved due to camera distortion.

The right image is the un-distorted version of the left one.



Compensating and correcting this distortion is relatively simple: we need some pictures of a chessboard:



Then, OpenCV has some functions to automatically recognize the coordinates of the corners of the chessboard. These points are then used to compute the parameters for camera calibration.

These parameters are then saved in a file, uploaded on the camera system, and used to un-distort camera images for face recognition.

3.3.11.8 Camera system endpoints

When the Python script runs, it listens on the port 5000. It exposes the following endpoints to TV:

- **[GET] http://camera_ip:port/start**

  The start actuation-command just tells the camera system to start the face recognition process.

- **[GET] http://camera_ip:port/stop**

  The stop actuation-command tells the camera system to stop the process, waiting for a new start command.

- **[POST] http://camera_ip:port/images**

  The images endpoint is for sending pictures and names of the known persons.

- **[DELETE] http://camera_ip:port/people/<name>**

  The people endpoint is for deleting pictures of the known persons by name (so it can delete more than one picture, if a name is associated to more pictures).

User doesn't need to use the camera system endpoints: these endpoints exist only for the TV.

3.3.12 CameraBot ThingVisor

The CameraBot ThingVisor is a ThingVisor able to interact with a robot in order to control its behaviour in terms of virtualization of its on-board video camera, exposing it as a set of configurable "static cameras". The moving robot sends a continuous stream of pictures back to the ThingVisor through a REST interface. The resulting pictures from the virtual set of static cameras, as well as constant updates about the timestamp and the sector the camera is pointing to, are available to VirIoT through the MQTT and HTTP data distribution systems of the platform.

The Thingvisor creates one Virtual Thing per static virtual camera, so that the robot is virtualized as an array of numbered cameras. Additionally, a virtual thing representing the robot itself, which is an actuator, so the user can send start/stop actuation-commands to it through the vSilo, is created.

The robot is fully autonomous, and it has sufficient intelligence to closely follow a closed path that is drawn on the floor by means of black tape, taking pictures at precise locations, meanwhile. Along the path, there are some QR Codes; each QR Code has a number, and represents a virtual camera pointing to that sector.

The robot must follow the path, by using the images coming from one of the on-board cameras to recognize the black line. It must also decode QR Codes; when a new QR Code is found, the robot takes a picture with a second camera and sends it to the TV through a REST request.

We chose the JetBot from Waveshare, because of the powerful capabilities of the NVIDIA JetSon Nano, and its out-of-the-box neural networks pre-trained for line following tasks.

3.3.12.1 Thingvisor parameters

The Camerabot TV accepts the following parameters:

- **num_of_cameras**: The number of cameras to configure. Some can remain empty, but you cannot get or fill non-existing cameras.
- **robot_ip**: The JetBot IP.
- **robot_port**: The port to which the JetBot running script is listening.
- **controller_url**: This is the URL of the master controller. It is necessary in order that the ThingVisor can configure automatically HTTP endpoints.
- **admin_id**: Username for logging in to the master controller.
- **admin_psw**: Password for logging in.

3.3.12.2 Virtual things

CamerBot ThingVisor creates a number virtual things:

- **robot**

  It's an abstraction of the robot itself, it's useful to send the start and stop actuation-commands.

- **camera0, camera1, . . . , cameraN-1** (where N is the number of camera)

  They represent the various virtual cameras along the path.

You can add the virtual things to a virtual silo, in order to receive messages or send actuation-commands.

3.3.12.3 NGSI-LD entities

NGSI-LD entities are produced whenever a new picture is avaible through the camera's HTTP entrypoint. Entities are as follows:

```
{
    "@context": [
        "https://uri.etsi.org/ngsi-ld/v1/ngsi-ld-core-context.jsonld"
    ],
    "id": "urn:ngsi-ld:CameraBotPicture:$",
    "type": "CameraBotPicture",
    "link_to_picture": {
        "type": "Property",
        "value": "/cameras/$/image"
    }
}
```

where $ is the id of the camera.

3.3.12.4 Sending actuation-commands to TV

One special vThing represents the Robot itself. When the Robot vThing is added to a vSilo, an NGSI-LD entity is created and sent to the broker. Let's look at the NGSI-LD entity that represents the Robot:

```
{
    "id": "urn:ngsi-ld:camerabot:robot",
    "type": "Robot",
    "commands": {"type": "Property", "value": ["start","stop","set_caching"]},
    "@context": [
        "https://uri.etsi.org/ngsi-ld/v1/ngsi-ld-core-context.jsonld"
    ]
}
```

For each actuation-command, the Silo controller creates the associated property. In order to send a actuation-command, modifying the associated property via the vSilo's broker is required.

Command values are JSON objects with the following properties:

- **cmd-value**: parameters of the command (mandatory)
- **cmd-qos**: required QoS (optional, default=0)
- **cmd-id**: unique command ID (mandatory)
- **cmd-nuri**: vSilo's unique notification URI, where to send NGSI-LD status/results

Whenever a user modifies a property representing a actuation-command, this is notified to the Silo controller, that will send it to the TV. The TV will execute the actuation-command, by doing whatever internal communication to the camera system is needed.

Here is a list of all available actuation-commands:

- **start**: Send the start actuation-command to the robot.
- **stop**: Send the stop actuation-command to the robot.
- **set_caching**: Enables or disables caching for a single camera.

The start actuation-command just tells the robot to start moving along the path, recognizing QR Codes and sending pictures. The stop actuation-command tells the robot to stop all activities, waiting for a new start actuation-command. If caching is disabled, the header "Cache-Control: no-cache" is sent every time the image is retrieved.

3.3.12.5 External HTTP endpoints for streaming pictures

Right after the ThingVisor starts up it talks to the Master Controller in order to automatically set up the HTTP endpoint to be used to reach the camera vThings' picture streams. The following HTTP base prefix is automatically configured, so as to allow vSilo's users to obtain access to all the external HTTP picture stream endpoints of the cameras, through the vSilo's sidecar HTTP interface:

`http://vsilo_ip:port/vstream/camerabot/cameraN` (where N is the index of the camera)

3.3.12.6 Internal endpoints to the robot

When the robot is started, we can pass IP and port of the ThingVisor as parameters. In this way, the robot can post to the TV. At the beginning, the robot presents itself to the TV, by sending its IP and port. However, this behaviour can be overriden by passing robot's IP and port directly to the TV as parameters when it's being created.

The CameraBot exposes the following REST entrypoints:

- **http://thingvisor_ip:port/init**

  At this time, init needs to be called once, in order to initialize the CameraBot mongo DB, by creating the various records for the virtual cameras, plus a record for the robot itself.

- **http://thingvisor_ip:port/robot/0**

  The robot can PATCH this entrypoint to send to ThingVisor its IP and port.

- **http://thingvisor_ip:port/cameras/N** (where N is the index of the camera)

  The robot can PATCH this entrypoint in order to update the current picture relative to the N-th camera. A GET to this entrypoint gives information about the specific camera, plus the current image.

- **http://thingvisor_ip:port/cameras/N/image**

  This GET entrypoint returns directly the image associated to the N-th camera.

3.3.12.7 Physical modifications to the Waveshare JetBot

Camera was tilted more to the floor, in order to avoid framing background objects in the room (that could hinder the neural network process).

The original motors didn't work well, because, below a certain power gain, they stopped suddenly. In other words, minimum robot speed was too high. So motors were swapped out with other compatible motors with higher torque, in order to run the robot slower.

Finally, a small led was added to light the floor, in order to capture lighter and higher quality images.

3.3.12.8 Software pipeline to control the JetBot

The software that runs on the JetBot is a script written in Python. The script initializes the main camera video stream with the following parameters:

- **Frame size**: 600 x 600 px
- **FPS**: 15

So, 15 times per second a callback is called. The image is first passed to a ResNet-18 neural network, trained to recognize the intersection point between the black line on the floor and an imaginary horizontal line in the frame. Comparing the position of this point with the center X coordinate, we get the orientation error.

Note: we use the torch2trt library to speed up the neural network, since speed is a key factor in systems with high inertia.

The error is passed to a properly calibrated PID controller, and the output is used to set the power of the two motors.

PID stands for proportional–integral–derivative controller, and is a control loop mechanism employing feedback, used in robotics and industrial control systems. In particular, the derivative component is really important in all systems with high inertia, because it allows to understand how the system is evolving, and so to anticipate the control action.

3.3.12.9 QR Code recognition

We use the same camera for the neural network and the QR Code recognition.

Since QR Codes needs to be sharp, we set the camera exposure time range properly (it's a synonym of shutter time). Too small times mean too noise, too high mean blurred pictures.

We use zbar-py Python library for QR Code scan. Since the main camera has an high FOV, image is distorted, and is not good for QR Code recognition. So we use OpenCV to undistort the image. Then a perspective correction is applied in order to make the floor top-down. In this way, QR Codes appear frontal, without any deformation except for rotation (but this one can be easily handled by the zbar-py library itself).

After these corrections, the QR Code can be recognized. QR Codes contain JSON objects with informations about the single locations (sector numbers) of the full closed path.

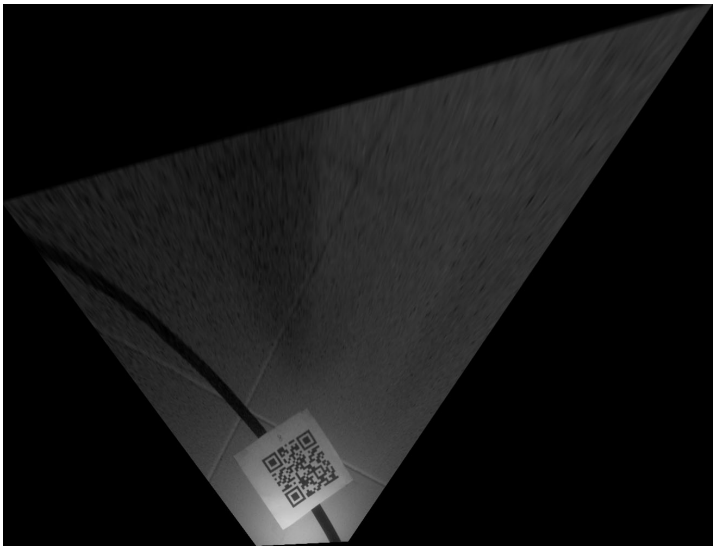
3.3.12.10 Camera calibration

This is the same as in paragraph 3.3.11.7

3.3.12.11 Perspective corrections

Here is an undistorted image of a QR Code:

Zbar (but also other libraries) doesn't work well with perspective-distorted QR Codes. So, before passing the image to Zbar, we do a perspective correction. The picture below is the perspective corrected version of the first one:



There is a small Python program to find the parameters for perspective correction. We just take a picture of a square piece of paper, then find the coordinate of the four corners; the program computes the matrix M to do the perspective warping.

QR Code recognition is rather expensive, so we run it in a separate thread, in order to try to not interfere with the neural network.

The main thread watches if the QR thread completed the QR Code recognition process. If yes, it sends to the QR thread the current image. So QR thread doesn't analyze all 15 images per second, but it's good enough to recognize a single QR Code more than once.

3.3.12.12 Test track

Figure 29 is a photo of a test track. The track's black line is made with electrical tape.

QR codes need to be flat and not creased, in order for the JetBot to read them. Moreover, currently a good external illumination is important for good recognition, since the led mounted on JetBot provides a non-uniform illumination, insufficient for QR codes recognition.

3.3.12.13 Results

Figure 30 is a diagram showing results and accuracy of the JetBot moving along the test track: the yellow line is the current path section number; every time the JetBot decodes a new QR Code, the number changes. So the JetBot traveled the path two times, passing two times through the zero starting point. The blue line is the error computed by neural network, while the red line is the PID output.

3.3.13 Philips Hue ThingVisor

This ThingVisor controls the lights connected to a Philips Hue bridge. It works both with a real bridge and with the Hue emulator (see `https://steveyo.github.io/Hue-Emulator`). It represents each light connected to the bridge as an NGSI-LD entity that has some properties that can be read and some other properties that correspond to actuation-commands used to act upon the lights, as in the following example:

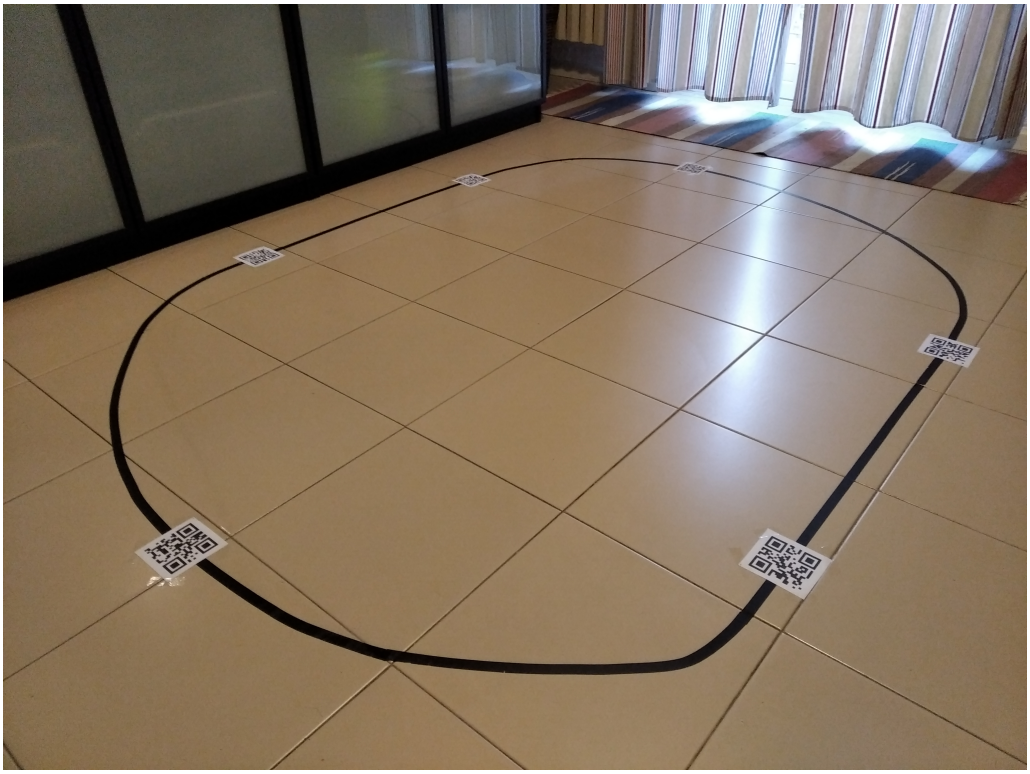


Figure 29: Sample track used for testing the CameraBot

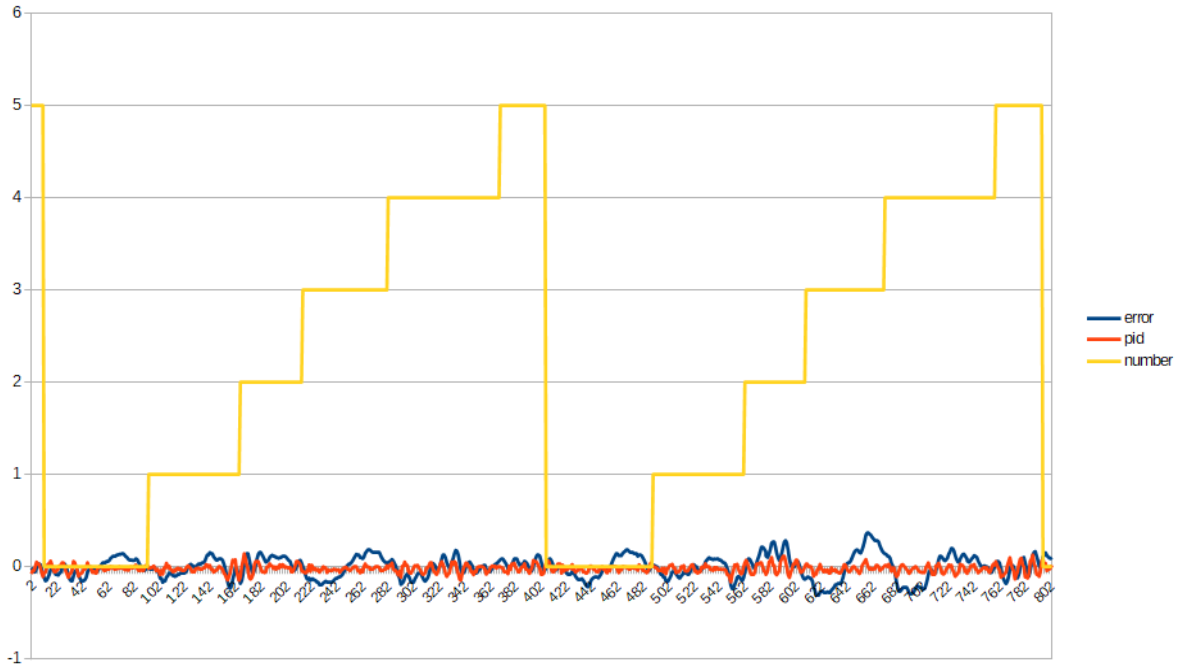


Figure 30: Accuracy of the CameraBot moving along the test track

```
{
 "id": "urn:ngsi-ld:phueactuator:light1",
 "type": "Extended color light",
 "brightness": {
  "type": "Property",
  "value": 254
 },
 "saturation": {
  "type": "Property",
  "value": 254
 },
 "hue": {
  "type": "Property",
  "value": 4444
 },
 "on": {
  "type": "Property",
  "value": true
 },
 "commands": {
  "type": "Property",
  "value": ["set-brightness", "set-saturation", "set-hue", "set-on", "raw-command"]
 }
}
```


Here is a list of all available actuation-commands:

- **set-brightness**

  The cmd-value of this actuation-command must be a number representing the desired brightness value of the light bulb.

- **set-saturation**

  The cmd-value of this actuation-command must be a number representing the desired saturation value of the light bulb.

- **set-hue**

  The cmd-value of this actuation-command must be a number representing the desired hue value of the light bulb.

- **set-on**

  The cmd-value of this actuation-command must be a boolean representing the on/off desired status of the light bulb.

- **raw-command**

  The cmd-value of this actuation-command must be a JSON object representing the desired custom actuation-command sequence to send to the light bulb, for instance:

  ```
  "cmd-value":{"transitiontime" : 30000, "on" : "True", "bri" : 254}
  ```

3.3.13.1 Thingvisor parameters

The Philips Hue TV accepts the following parameters:

- **bridgeIP**: the IP address of the bridge (172.17.0.1 for Hue Emulator)
- **bridgePort**: the port of the bridge (e.g. 5000 for the emulator, 80 for a real Hue bridge)

To connect the ThingVisor with the Philips Hue bridge, pressing the Bridge Button either on the emulator or a real bridge and waiting few seconds is required. Checking that the connection is in place can be done by monitoring the status of the ThingVisor, through the `list-thingvisors` CLI command.

3.3.14 LoRaWAN ThingVisor

This ThingVisor helps to connect vThings to real devices living in a LoRaWAN network managed by a Chirpstack server.

Each vThing created by this ThingVisor is associated to a LoRaWAN device, it is thus identified using the Chirstack application id (appid) and the deveui of the device; for the thingVisor to properly integrate it in the Viriot platform, we require also a label and a type of device. This latter parameter has a predefined set of values, which tells to the thingVisor which data structures are used into the communications and how they are translated into NGSI-LD. For now, the only supported type for a device is *smartcam*; but this can be easily extended.

The list of vThings is given in the configuration at ThingVisor startup

Example configuration to connect a LoRaWAN ThingVisor to two smart cameras

```
{
  'chirpstack_mqtt_server':'example.com',
  'chirpstack_mqtt_port':'8883',
  'chirpstack_cafile' : (content of base64 -w0 '/path/to/ca.crt') ,
  'chirpstack_crtfile': (content of base64 -w0 '/path/to/user.crt'),
  'chirpstack_keyfile': (content of base64 -w0 '/path/to/user.key') ,
  'devices'=[
    { 'type':'smartcam',
      'label':'SC-ABC',
      'appid':'12',
      'deveui':'0123456789ABCDEF'
    },
    { 'type':'smartcam',
      'label':'SC-DEF',
      'appid':'12',
      'deveui':'ABFE356789ABCDEF'}
  ]
}
```

When the ThingVisor initializes, it subscribes to the related events on the MQTT broker of the Chirpstack server and do the needed data translation.

3.3.15 FogFlow ThingVisor

This component serves as the bridge between distributed ThingVisors that are constructed using the FogFlow ThingVisor Factory and VirIoT. See Section 4.1.1.

3.4 Developed Virtual Silo Flavours

vSilos are IoT Infrastructures offered as a service. Tenants can already choose among a variety of vSilos flavours (images of vSilos), but adding new flavours is possible.

As shown in Figure 3, every vSilo is composed of four main building blocks:

- the vSilo IoT Broker. This is an IoT Broker (such as FIWARE's Orion, oneM2M's Mobius, NEC's Scorpio, a Mosquitto server, etc...) that is incorporated into the vSilo and serves data coming from the Virtual Things (that have been added to the vSilo) to the external IoT Application.
- the vSilo IoT Controller. It is a custom software piece that: i) translates NGSI-LD context data of Virtual Things (i.e. ThingVisors) to the information model of the IoT Broker (see D4.2), ii) handles VirIoT Control Procedures. The communication channel between the IoT Controller and Broker as well as between the Broker and the external Applications uses the Broker API and is usually either RESTful or pub/sub based (HTTP, MQTT, CoAP).
- the vSilo HTTP Broker. This is a kind of HTTP proxy incorporated into the vSilo and serves HTTP GET/POST requests of generic contents offered by vThings, which come from external Applications .
- the vSilo HTTP Controller. It is a software piece that is able to manage VirIoT control procedures related to the control of HTTP services.

Table 8 summarizes the vSilos the Fed4IoT project has developed so far. The following sections give details about operation and implementation of each one of them. Currently, the HTTP services of the developed vSilo are always offered by a HTTP sidecar container described in Section 3.4.1.

Name	IoT Broker	IoT Information Model	Broker Developer	Reference
Mobius oneM2M	Mobius	oneM2M	OCEAN open alliance	`https://github.com/fed4iot/VirIoT/tree/master/Flavours/mobius2-pub-sub-actuator-flavour`
NGSI-LD	Scorpio, OrionLD, Stellio	NGSI-LD	NEC, FIWARE, EGM	`https://github.com/fed4iot/VirIoT/tree/master/Flavours/ngsi-ld-flavours`
Orion NGSIv2	Orion	NGSIv2	FIWARE	`https://github.com/fed4iot/VirIoT/tree/master/Flavours/orion-flavour`
Mosquitto Raw JSON	Mosquitto	JSON	Eclipse Foundation	`https://github.com/fed4iot/VirIoT/tree/master/Flavours/raw-mqtt-actuator-flavour`

HTTP vSilo side-car	-	-	CNIT	https://github.com/fed4iot/VirIoT/tree/master/Flavours/http-sidecar-flavour

Table 8: vSilo Flavours Portfolio

3.4.1 HTTP vSilo sidecar

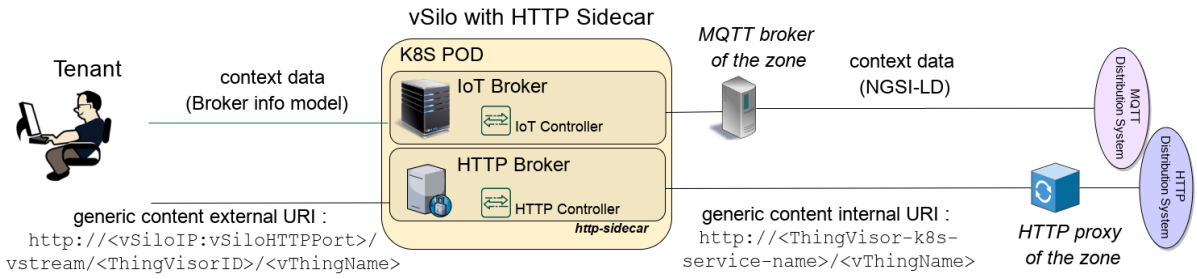


Figure 31: HTTP vSilo Sidecar

The HTTP vSilo Sidecar is not actually a stand-alone vSilo but is a container that can be added to any vSilo without native HTTP support to add it to them. Currently, any developed vSilo uses this sidecar to offer HTTP services. Consequently, in what follows, we only present their context data related functionality.

Figure 31 shows the architecture of a vSilo that includes the HTTP sidecar. The HTTP sidecar is a process that expose an HTTP external endpoint where it receives HTTP GET/POST requests from the Tenant. The external URI is `http://<vSiloIP:vSiloHTTPPort>/vstream/<ThingVisorID>/<vThingName>`. The HTTP Broker included in the sidecar is a kind of proxy that receives these requests, changes the URI to the internal URI exposed by the vThing, i.e. `http://<ThingVisor-K8S-service-name>/<vThingName>`, and forwards this GET/POST to the proxy of the HTTP Distribution System located in its VirIoT zone. From here, the request will be forwarded to the ThingVisor that will reply with the requested content. The HTTP controller included in the sidecar, handles the control procedures related to the adding and removal of a vThing to the vSilo and in turns enables the HTTP Broker to forward (if vThing is in the vSilo) or reject (vice-versa) HTTP GET/POST requests for the vThing generic contents.

3.4.2 Orion NGSIv2 Flavour

This vSilo flavour contains uses a NGSIv2 Orion Context Broker as IoT Broker. The vSilo IoT Controller receives NGSI-LD context data of the vThings through the internal MQTT Distribution System. The vSilo IoT Controller processes the NGSI-LD payload and builds the corresponding payload in NGSIv2 format using a wrapping tool, following the mapping rules specified in D4.2. After performing this translation task, the vSilo IoT Controller can store the context data into the NGSIv2 internal Orion Context Broker using the NGSIv2 RESTful API.

This vSilo flavour also supports the control of vThing that are virtual actuators controller by actuation-commands, as specified in D3.3 and D4.2. In this sense, Figure 32

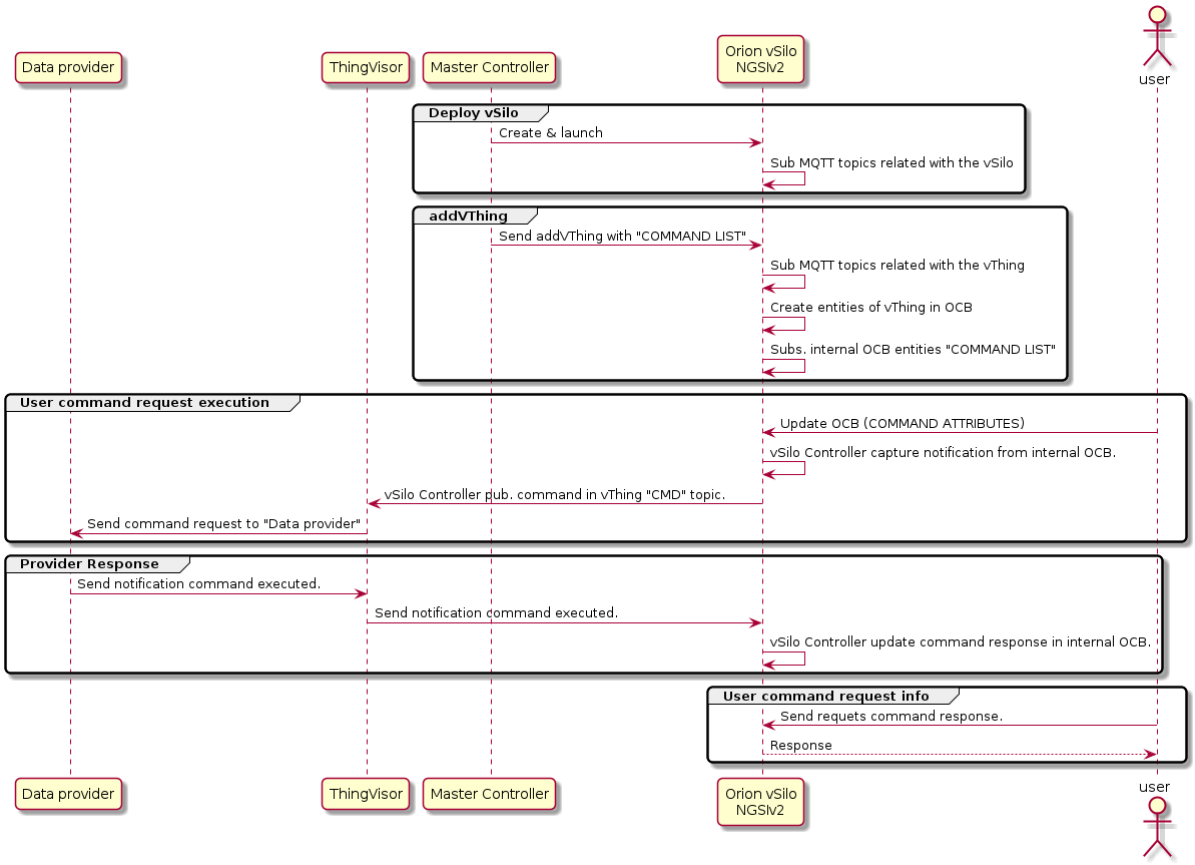


Figure 32: vSilo Orion supporting actuation-commands

shows how the vSilo IoT Controller catches the list of possible actuation-commands corresponding to the added vThings which are virtual actuators. After that, related actuation-command pipes are added as Thing' attributes to the Orion Context Broker, and the vSilo IoT Controller subscribes to these attributes. When a user updates one of these attributes to send an actuation-command, a notification arrives at vSilo IoT Controller which sends the actuation-command upstream to the vThing through the MQTT Distribution System. Finally, the vSilo IoT Controller waits for the command response and updates the Orion Context Broker when receives it. This pub/sub modus-operandi between IoT conteollr and Broker for the management of actuation-commands is also used by the other vSilos presented in this document, although in the case of oneM2M/Mobius vSilo the command pipes are oneM2M containers, in the case of NGSIv2 vSilos the command pipes are Entity Properties, and in the case of the raw MQTT/Mosquitto vSilo the command pipes are simply MQTT topics.

3.4.3 Mobius oneM2M Flavour

This vSilo Flavour contains a vSilo IoT Controller that is able to transform NGSI-LD data and metadata into oneM2M data and metadata. It performs a straightforward mapping between NGSI-LD and oneM2M along the guidelines for automatic translation that we have defined in D4.2. The IoT Broker of the vSilo is a Mobius (v2) server, which

is the open source IoT server platform based on the oneM2M standard that has received certification by TTA (Telecommunications Technology Association). This vSilo flavour also supports the control of vThing that are virtual actuators as specified in D3.3 and D4.2.

3.4.4 NGSI-LD Flavour

The generic NGSI-LD vSilo can have any NGSI-LD-compatible broker as IoT Broker and its IoT Controller is able to replicate upstream NGSI-LD data to the broker. Specifically, it has the ability to map, into NGSI-LD, both data and metadata by exploiting NGSI-LD Relationships between Entities and vThings. This approach is a consequence of our information model design, which is also meant to support discovery and indexing of vThings inside the VirIoT platform, in general. More details can be found in deliverable D4.2. This vSilo flavour also supports the control of vThing that are virtual actuators as specified in D3.3 and D4.2. Scorpio is the NGSI-LD-capable Context Broker under active development by NEC. OrionLD is the NGSI-LD-capable Context Broker under active development by FIWARE. Stellio is the NGSI-LD-capable Context Broker under active development by EGM.

3.4.5 Mosquitto Raw JSON Flavour

This virtual silo flavour is designed so as to export the IoT data coming from its Virtual Things via simple MQTT topics. This is a kind of *raw* Virtual Silo, which can be in turn connected to an upstream IoT platform such as Node-Red or Google/Azure/Amazon IoT cloud services, according to the application design and deployment strategies. Specifically, this vSilo incorporates a Mosquitto server instance as IoT Broker. Upon receiving data, the vSilo IoT Controller simply publishes any data payload to a Mosquitto topic that is named according to the tenant name and the vThing data is coming from: `tenant_id/v_thing_id`. Any Application can then subscribe to this tenant-specific topic, and can receive a raw copy of context data and metadata.

4 ThingVisor Advanced Orchestration and Development Tools

4.1 FogFlow-based ThingVisor Factory

This section introduces the high level system overview of FogFlow and its advanced programming model and service orchestration mechanisms that have been used to support the development and management of advanced ThingVisors in a cloud-edge environment. Later, it describes how FogFlow is integrated with the other Fed4IoT components to realize the FogFlow-based ThingVisor Factory, and then it reports some performance results. More detailed information on how to use FogFlow is provided by the FogFlow online tutorial [3].

4.1.1 Concept

In VirIoT, FogFlow is used as an advanced ThingVisor Factory, able to implement distributed ThingVisors that dynamically orchestrate various vThings and span from cloud to edge, as illustrated in Figure 33. In the latest version of FogFlow, developed during the Fed4IoT project, NGSI-LD is now used as the internal data model and communication protocol to exchange information between different vThings (it was NGSIv1 previously), thus eliminating any translation overhead when data leaves the distributed ThingVisor towards other VirIoT components. This FogFlow-based distributed ThingVisor thus exposes its vThings to VirIoT through a transparent bridge ThingVisor component (see Section 3.3.15). FogFlow is able to orchestrate multiple vThings both in the cloud and at the edges, making the vThings closer to their physical counterparts in the Root Data Domain. More importantly, FogFlow allows vThings to interact with each other directly at the edge when the corresponding things are close to each other. Overall, it offers three main technical benefits:

1. reduce the internal bandwidth consumption and communication latency between vThings and their physical things counterparts;
2. enable *direct* communication and interactions between vThings, seamlessly over the cloud and edges;
3. provide the flexibility and *programmability* to realize different ThingVisors with its intent-based edge programming model.

Within FogFlow, each physical thing is represented as an NGSI-LD entity associated with a set of data services that are programmed and orchestrated based on the FogFlow programming model. Currently, FogFlow supports the following three types of data services around a virtual thing.

- **Synchronization service** to enable the bi-directional data transformation and synchronization between a virtual thing and its corresponding physical thing. It is responsible for dealing with the heterogeneity and interoperability of communication protocol and data model on both sides.

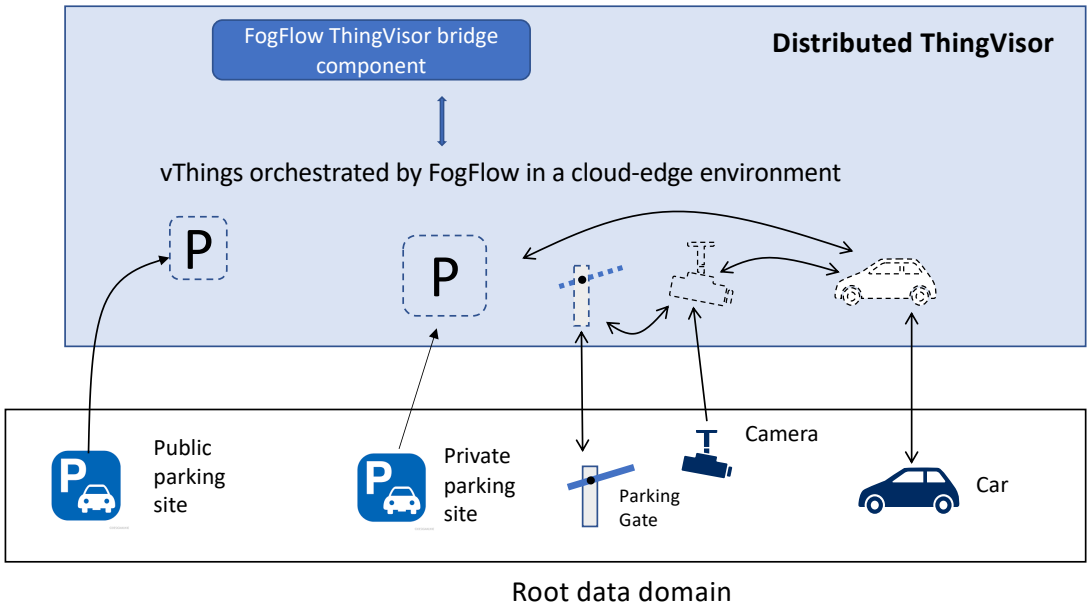


Figure 33: Virtual Things Located at Cloud and Edges

- **Internal atomic service** to take some existing properties as inputs to perform some data analytics and then use the generated analytics result to create or update a new property for the same virtual thing.
- **External atomic service** to take the input data from other virtual things to perform some data analytics and then use the generated analytics result to create or update a new property for the local virtual thing.

A concrete example of these three services can be seen from Figure 34 in a scenario related to smart parking. In such a scenario, when a connected car is entering a parking house with a connected gate controller and a connected camera, two virtual things are created, one for the gate controller with the MQTT interface connected to a MQTT broker running at the edge and the other for the camera with oneM2M interface connected to a oneM2M gateway running at the edge.

Two services are orchestrated for the virtual camera: 1) a synchronization service that can fetch the oneM2M data and use the converted information to update the NGSI-LD based data presentation of the virtual camera; 2) an internal service called "Car-plate detection" that can take the "STREAM_URL" of the camera as the input and then constantly read its video stream to perform real-time car plate detection and update the "DETECTED_CAR" property with the detected car plate number.

Another two services are also orchestrated for the virtual gate controller: 1) a synchronization service that can fetch the MQTT message reported by the gate controller and use the converted information to update the NGSI-LD based data presentation of the virtual gate controller; in the meantime the synchronization service will also subscribe the updates of the "GATE_STATUS" property of the virtual gate controller and then write them back to the gate controller as command messages; 2) an external service called "plate validation" that can subscribe to the detected car plate number from the virtual camera and then update the "GATE_STATUS" property of the virtual gate

controller after a plate validation procedure. This example shows that, with the help of FogFlow, a complex control process between two physical things can be accomplished fast and automatically via the communication/interaction between their synchronized virtual things.

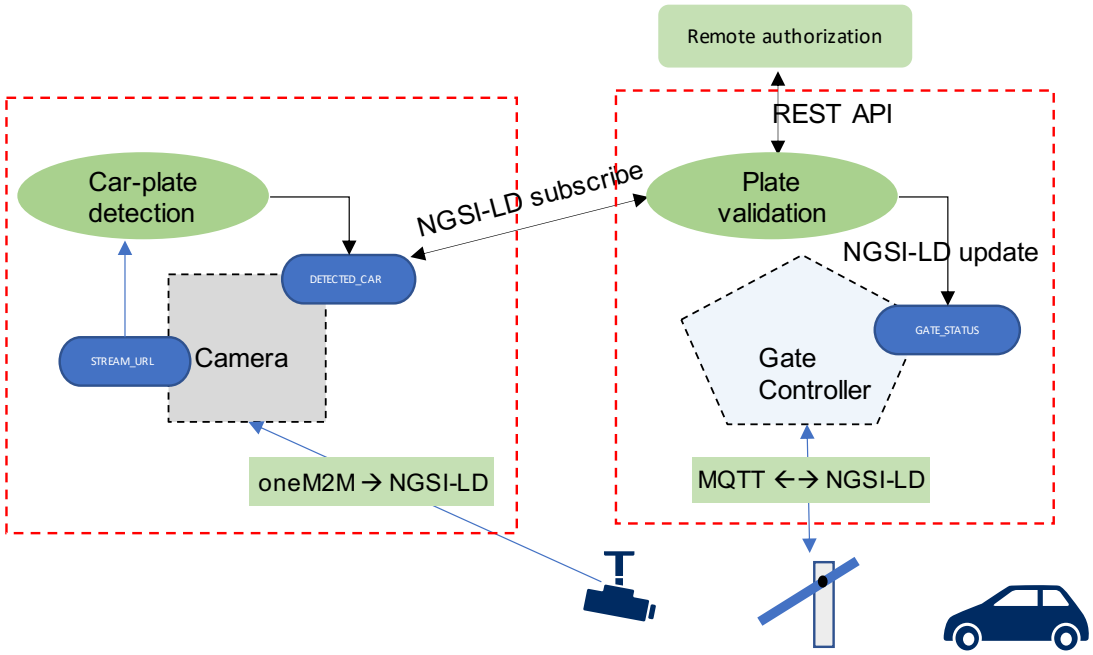


Figure 34: FogFlow ThingVisor Applications

4.1.2 Intent-based Programming Model in FogFlow

As illustrated by Figure 35, in FogFlow an IoT data service is represented by a service topology and a set of user-defined intents.

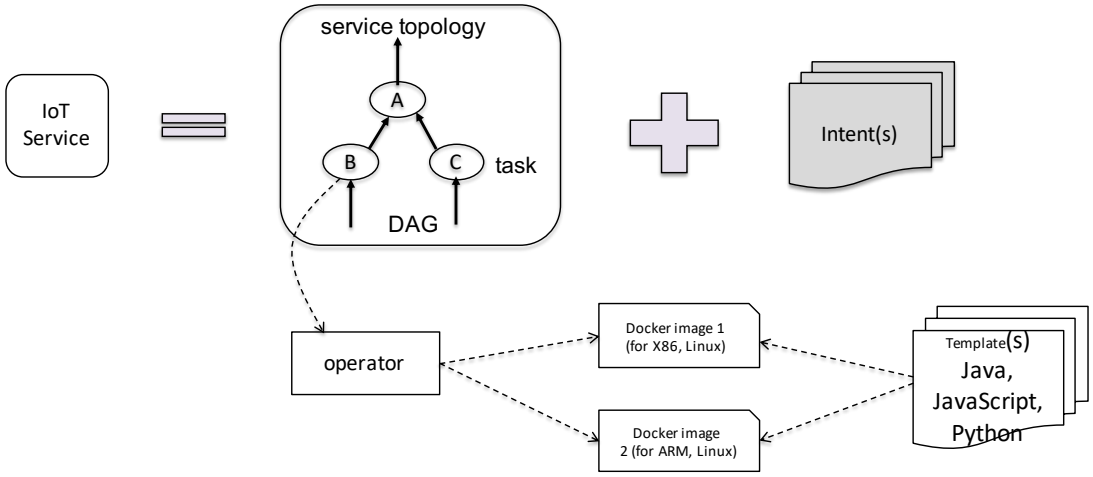


Figure 35: Service Model in FogFlow

The service topology is a graph of tasks, and each of them is supposed to perform some type of data processing. Tasks in the same topology are linked with each other,

based on the dependency of their data inputs and outputs. Each task is annotated by service designers via a graphical editor, to define their input and output data and to also define a granularity feature that determines how input data should be divided into task instances, for parallelization of computation. Each task instance runs within a docker container. By design, a service topology only defines the data processing logic of an IoT service.

To trigger the service topology in FogFlow, service consumers need to define an intent to express their high-level goals of using such as IoT service. More specifically, as illustrated by Figure 36, an intent can be customized to cover the following goals:

1. *service topology* that defines which service logic to be triggered;
2. *geoscope* that defines the scope to select the input data for applying the selected service topology;
3. *service level objective (SLO)* that defines the service level objective to be achieved, in terms of latency requirements, bandwidth saving, or privacy/security needs;
4. *priority* that defines how the triggered service deployment could utilize the shared infrastructure resources with the other existing services.

With such an intent-based programming model, FogFlow is able to dynamically orchestrate concrete service deployment plans to meet any user-defined intents in a more flexible way, even for the same service topology.

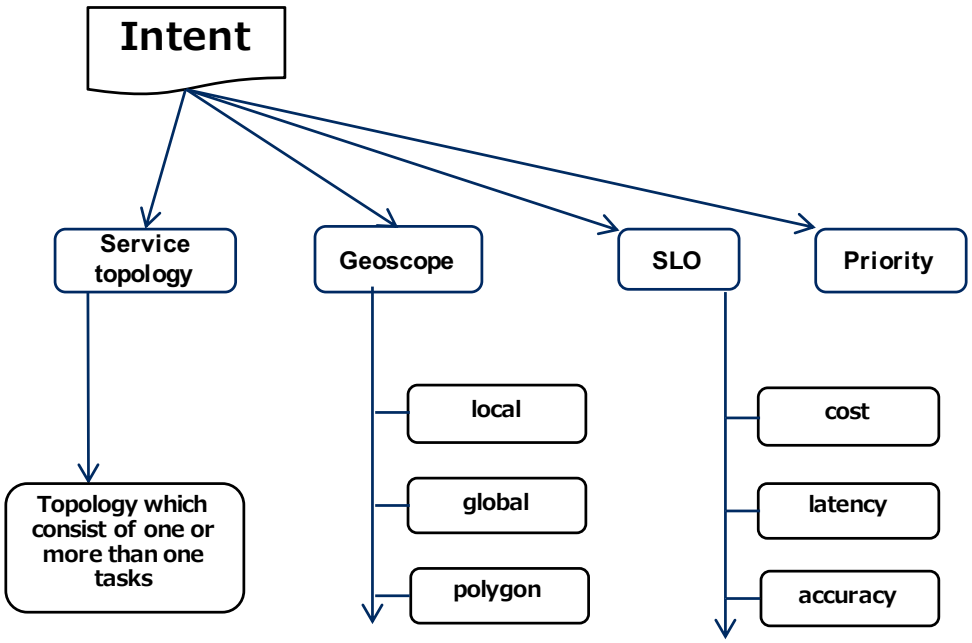


Figure 36: Intent Model in FogFlow

As shown in Figure 37, programming a FogFlow service includes three key elements: *operator*, *service topology*, and *intent*. An operator represents a type of data processing unit, for example, calculating the average temperature, performance the face matching of

two face images. A service topology represents the computation logic of the IoT service and consists of several linked operators annotated with their data inputs and outputs. An intent is a JSON object that follows the proposed intent model to define a customized requirement of how the service topology should be triggered.

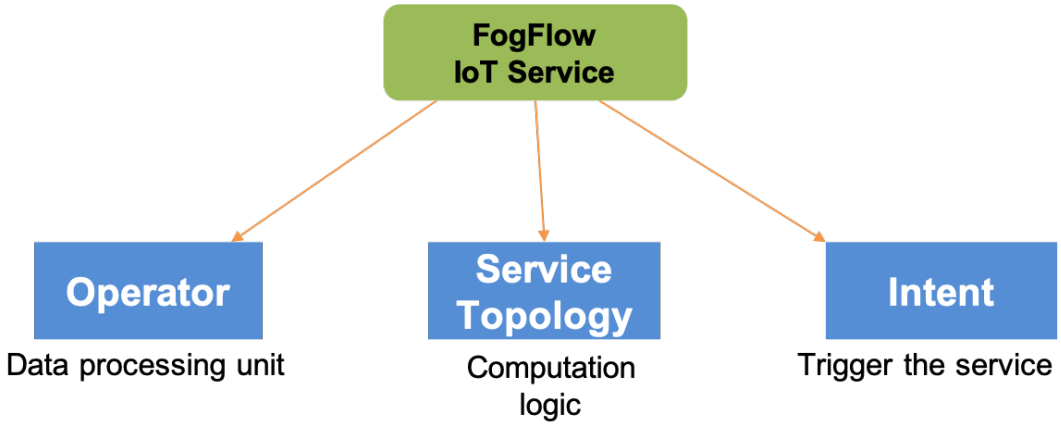


Figure 37: Three key elements to program an IoT service in FogFlow

Currently, FogFlow can also support serverless fog computing by providing so-called Fog Function, which is a common easy case of the intent-based programming model. As illustrated in Figure 38, Fog Function represents a common special case of the generic intent-based programming model in FogFlow, meaning that a fog function is associated with a simple service topology that includes only one task (a task is mapped to an operator in FogFlow) and an default intent that takes "global" as its geoscope. Therefore, when a fog function is submitted, its service topology will be triggered immediately once its required input data is available.

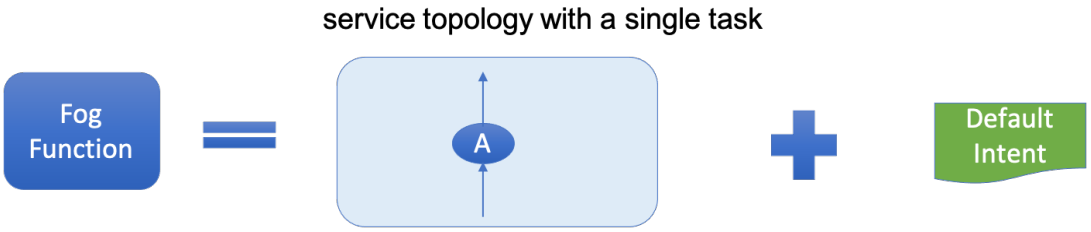


Figure 38: FogFunction as a simple case of service topology in FogFlow

4.1.3 FogFlow System Framework

FogFlow is an open source fog computing framework that can dynamically orchestrate IoT services over cloud and edges on-demand, in order to fulfill high-level "service intention" expressed by service consumers, which could be external applications or any IoT devices. Figure 39 shows a high level view of the FogFlow system. It consists of a number of *fog nodes*, each of which runs a *Broker* and a *Worker*. A management node runs two centralized components, namely *Discovery* and *Orchestrator*. Each node is a Virtual Machine (VM) or physical host deployed either in the cloud or at edges. All fog nodes

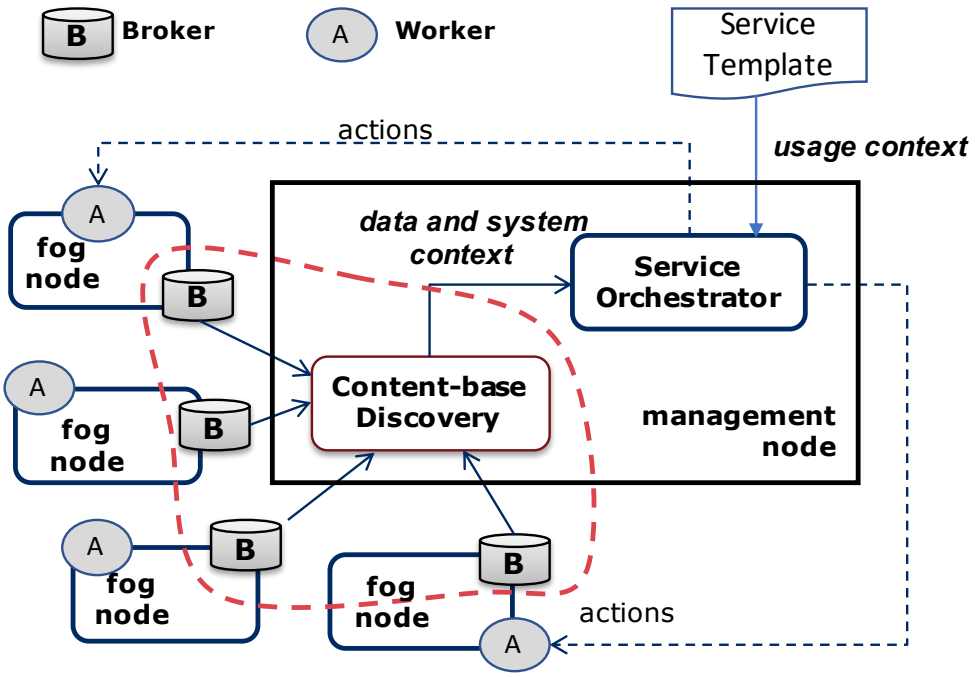


Figure 39: System Overview of FogFlow

form a hierarchical overlay based on their configured GeoHash IDs. All data in the system is represented as entities saved by a Broker and indexed by the centralized Discovery for discovery purposes. The data can be raw data published by IoT devices, intermediate results generated by some running data-processing tasks, or data available at a resource, reported by fog nodes. When a fog function is registered, Orchestrator will subscribe to the input data of the fog function to Discovery. Once the subscribed data pieces appear or disappear in the system, Orchestrator will be informed, and it can then take orchestration actions accordingly, which will be carried out by an assigned worker.

Figure 40 shows the user interface implemented by FogFlow to allow service developers to trigger a service topology on the fly. Figure 41 shows the user interface implemented by FogFlow to allow service developers to trigger a service topology on the fly.

4.1.4 Context Aware Service Orchestration

Another unique feature of FogFlow is context aware service orchestration, meaning that FogFlow is able to orchestrate dynamic data processing flows over cloud and edges based on the following three types of contexts, including:

Data context: the structure and registered metadata of available data, including both raw sensor data and intermediate data. Based on the standardized and unified data model and communication interface, namely NGSI, our system is able to see the content of all data generated by sensors and data processing tasks in the system, such as data type, attributes, registered metadata, relations, and geo-locations.

System context: available resources at each fog node. The resources in a cloud-edge environment are geo-distributed and they are dynamically changing over time. As

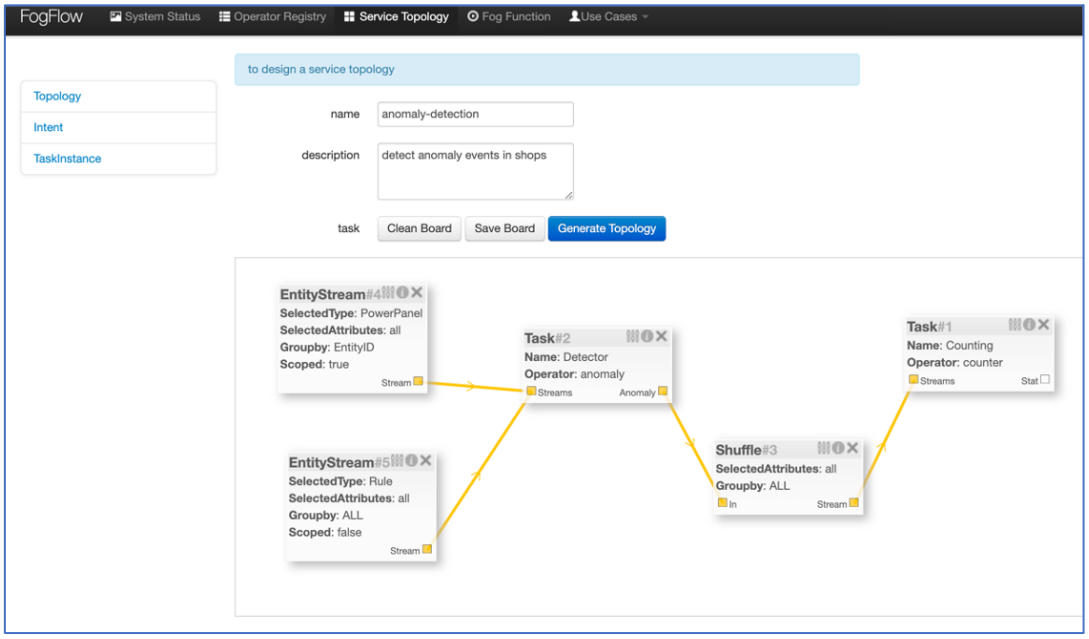


Figure 40: Editor of service topology in FogFlow

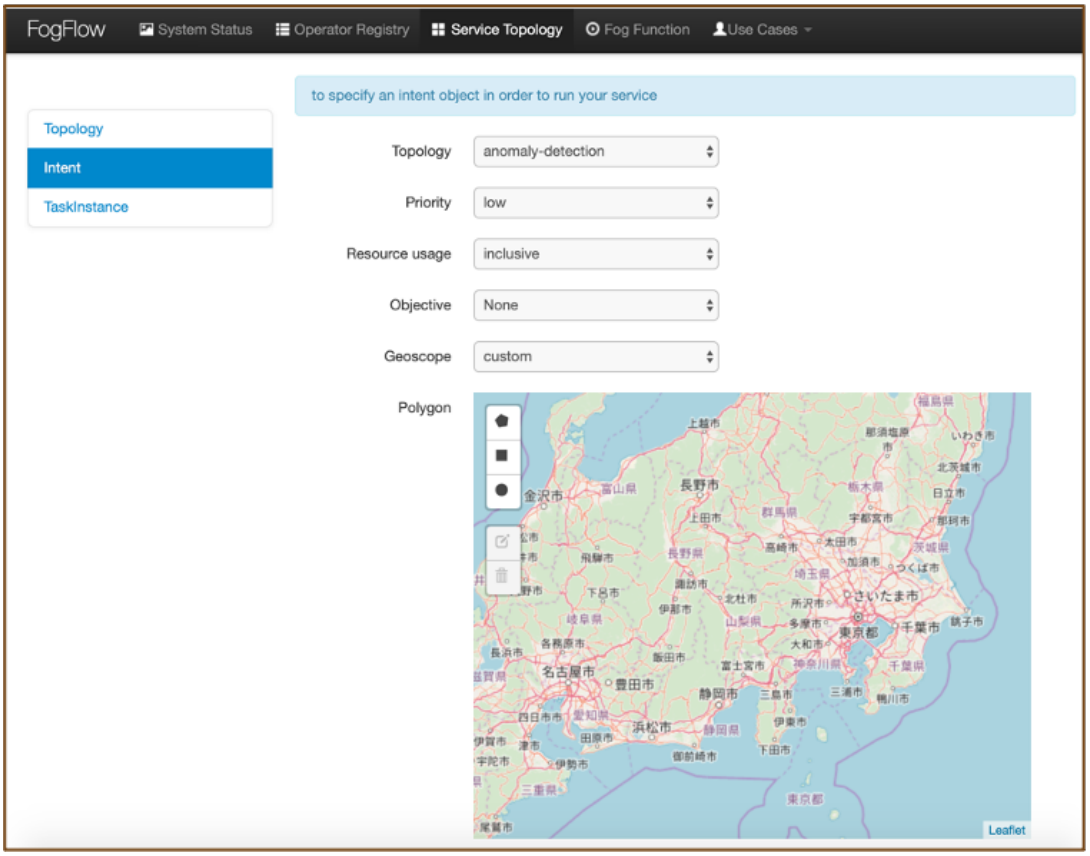


Figure 41: User interface to define an intent in FogFlow

compared to cloud computing, resources in such a cloud-edge environment are more heterogeneous and dynamic.

Usage context: high level usage intentions defined by service designers to indicate what their fog functions should be used in the system, such as which type of results is expected under which type of QoS within which geo-scope.

Figure 42 shows the major procedure for Orchestrator to orchestrate fog functions based on the update notification of context availability of their input data, provided by Content-based Discovery. More specifically, the following four basic orchestration actions are designed to dynamically orchestrate tasks for each registered fog function.

- *ADD_TASK*: To launch a new task with the given configuration that includes the initial setting of its input streams. When launching a new task, the Worker first fetches the Docker image for this task and then launches and configures this task within a dedicated Docker container. After that, the Worker subscribes the input entity to the context management system on behalf of the running task so that the input streams can be received by the running task; in the end, the newly created task is reported back to the orchestrator.
- *REMOVE_TASK*: To terminate an existing running task with the given task ID. When terminating an existing task, the Worker not only stops and removes its corresponding Docker container, but also unsubscribes its input streams so that the context management system does not end up with lots of unavailable subscribers.
- *ADD_INPUT*: To subscribe to a new input stream on behalf of a running task so that the new input stream can flow into the running task.
- *REMOVE_INPUT*: To unsubscribe from some existing input stream on behalf of a running task so that the task stops receiving entity updates from this input stream.

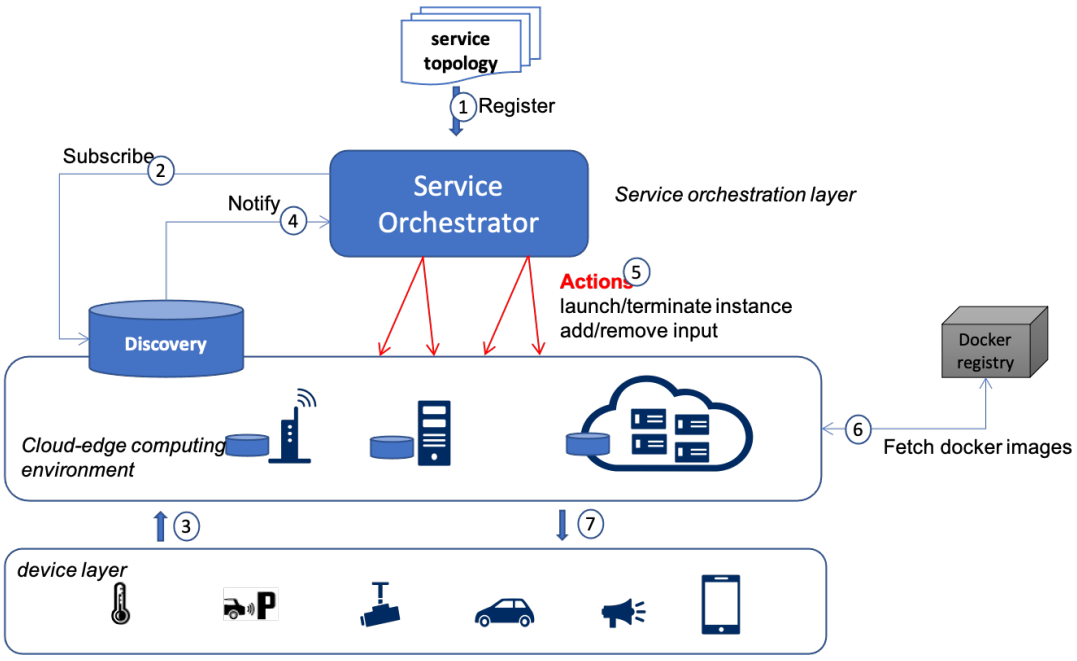


Figure 42: Data-driven orchestration

4.1.5 Decentralized Orchestration

The initial version of FogFlow provides only the centralized implementation of Discovery and Service Orchestrator. However, this centralized approach has limited scalability and reliability and also leads to a long delay of launching a new data service on the fly. This is because, with the centralized approach, all orchestration decisions must be made by the centralized service orchestrator, which is usually deployed on the cloud node of the FogFlow system. To avoid this bottleneck and also improve the scalability and reliability of the service orchestrator in FogFlow, the latest FogFlow introduces a new decentralized orchestration mechanism based on distributed discovery and orchestrator, as illustrated in Figure 43. The key idea behind the decentralized orchestration is to leverage of the geoscope information associated with intents to break down the entire orchestration workload into different regions, allowing each orchestrator that runs at each edge node or in the cloud to make its own orchestrations locally.

With the decentralized service orchestration, the entire FogFlow system consists of a set of autonomous agents and each of them includes the same set of components, worker, broker, discovery, and orchestrator, but they are organized at different logical layers to cover different regions of the entire geomap. A global routing table that indexes which agent is responsible for which region is propagated and maintained via a gossip protocol among all agents. As shown in Figure 36, every intent to trigger the service deployment will be associated with a pre-defined geoscope, which indicates which regions are covered or overlapped with this intent and then can be used to look up which agents should be involved to deal with this intent. This is all done via the distributed discovery and broker network across agents. In the end, this intent will be forwarded to all involved orchestrators and each of them will start to monitor the context information in its own local region and then make the proper orchestration decisions locally and immediately to deploy new tasks whenever the required input data becomes available from its own broker.

4.1.6 FogFlow-based ThingVisor Factory

During the Fed4IoT project, FogFlow has been extended on two major aspects. First, the internal data model and communication protocol in FogFlow have been changed from NGSIv1 to NGSI-LD, so that the communication/interactions between two vThings can be carried out efficiently at the semantic level by referring to the same vocabulary. This is important especially when these two virtual things are from two different root data domains managed by different organizations. Second, as illustrated in Figure 44, a new module called ThingVisor Factory is designed and implemented in FogFlow to create the NGSI-LD based data representation of virtual things and also to manage all data services around the vThings. For each type of vThing, the FogFlow-based ThingVisor Factory requires two types of inputs.

- *Thing Class*, which is a class to define all common properties for this type of virtual things and also the associated data services to create/update those properties;
- *Thing Profile*, which provides the identity and configuration information of an individual thing, such as its unique ID, entity type, geo-location, accessible URL of its root data domain.

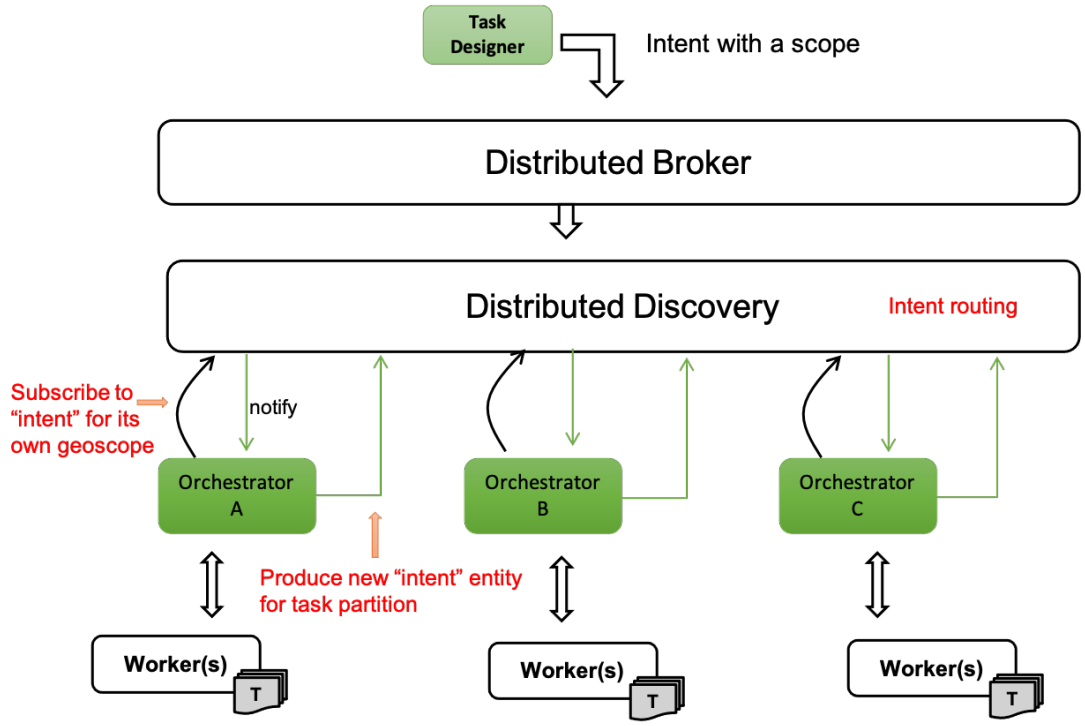


Figure 43: Decentralized service orchestration with scoped intent

By taking these two types of inputs, the FogFlow ThingVisor Factory can translate them into the corresponding intents, which are further interpreted by the FogFlow service orchestrator to configure and deploy all data services (synchronization service, internal service, and external service) defined in the ThingVisor Class. Currently, FogFlow provides a graphical user interface to specify ThingVisor Classes, a programming model to program data services, and also a REST-based interface to create/update Thing Profiles.

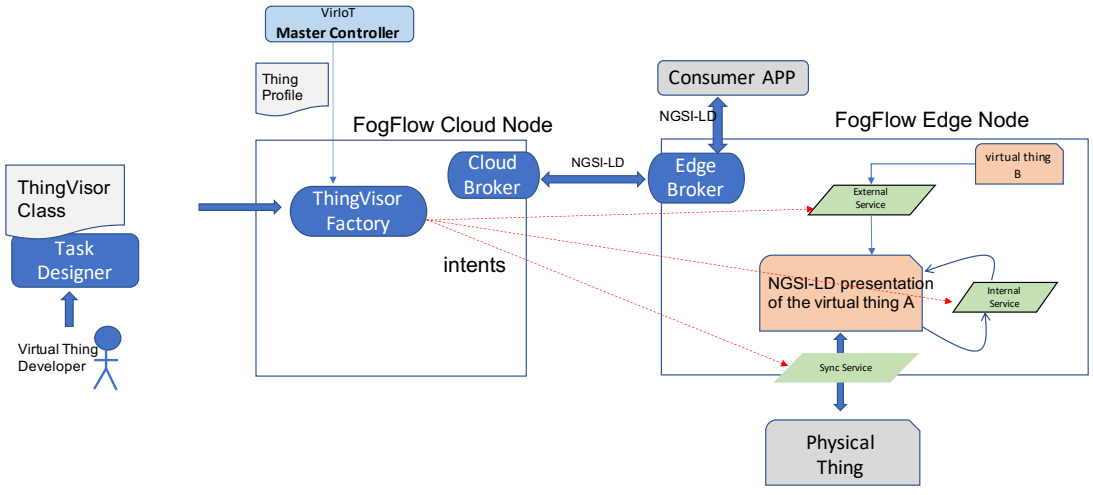


Figure 44: FogFlow ThingVisor Factory

With the intent-based edge programming model in FogFlow, we can implement different types of ThingVisors in a more flexible way. For example, we can implement a

very specific ThingVisor that is used to create only one specific Virtual Thing; however, we can also implement some generic ThingVisors that can be used to create a number of Virtual Things of the same kind, on-the-fly for a given geo-scope. In addition, with the fog function programming model, we can implement some ThingVisors that can create Virtual Things automatically.

Figure 45 shows how FogFlow works with the other VirIoT components. FogFlow can be used to implement different (distributed) types of FogFlow-based ThingVisors, each of which is implemented as a FogFlow service based on the intent-based programming model. Assume that those FogFlow-based ThingVisors are developed and registered in FogFlow via the GUI of FogFlow Task Designer. Then those ThingVisors can be managed by the (`/addThingVisor` and `/deleteThingVisor`) VirIoT APIs, via a generic FogFlow-ThingVisor bridge component, which is a dockerized application to communicate with the running FogFlow system, for managing the specific FogFlow-based ThingVisors. Since each FogFlow-based ThingVisor is implemented as a FogFlow service, adding or deleting a ThingVisor is equal to enabling or disabling a FogFlow service via a customized intent.

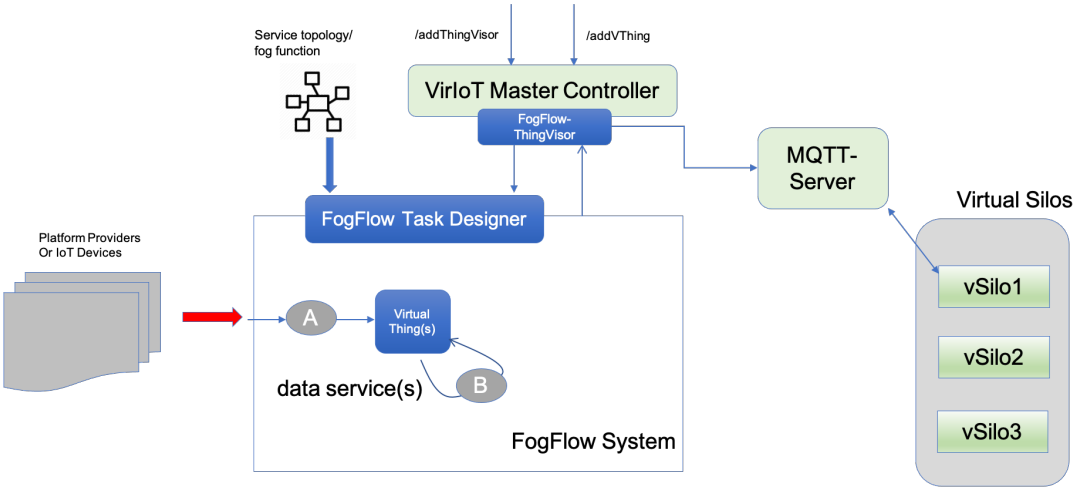


Figure 45: FogFlow in VirIoT

4.1.7 Performance Evaluation

The following experiments have been carried out to evaluate the performance and scalability of the FogFlow system with three fog nodes from Google Cloud, located at its data centers in three different regions (Frankfurt, Ireland, and Tokyo). We also compare the performance of the centralized service orchestration and decentralized orchestration.

Figure 46 shows how the throughput of creating new NGSI entities changes as the number of fog nodes increases from 1 to 3. This stress test is done with JMeter, which can issue continuous entity creation requests from 100 threads simultaneously. The same test is carried out both for the centralized orchestration and for the decentralized orchestration. Overall, as the number of fog node increases, the aggregated throughput from all fog nodes increases as well. However, the throughput with the decentralized orchestration is higher and shows a linear increase pattern. This is mainly because, with the decentralized orchestration, the entity creation workload can be well handled by the local

broker and discovery inside each FogFlow agent. With the centralized orchestration, the discovery component is still centralized and deployed on one fog node. When the brokers on the other two fog nodes create new entities, they always have to update this centralized discovery. This could slow down the entity creation throughput on the other two fog nodes.

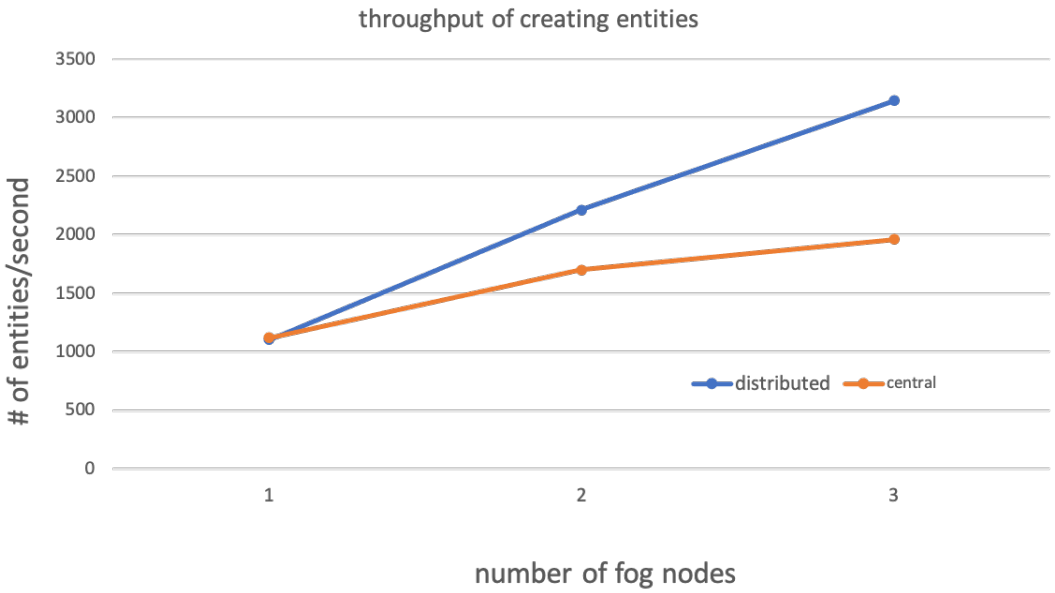


Figure 46: Throughput of creating new entities in FogFlow

Figure 47 shows the performance of the service orchestrator(s) to launch new tasks for an intent. In the test, the intent is specified with the global scope to create a virtual thing for each individual temperature sensor. In order to inspect how fast the centralized and decentralized service orchestrators can make their orchestration decisions to launch new tasks, we actually disable the step to spawn the orchestrated tasks in a docker container in the test. This step is known to be the major bottleneck for launching tasks in FogFlow, but this can be largely improved by using Unikernel, instead of docker container. As shown in Figure 47, as the number of fog nodes increases, more new tasks can be orchestrated per second. This is true for both the centralized orchestration and decentralized orchestration. But clearly, the decentralized orchestration can scale much better than the centralized approach, almost with a linear scale-up. Also, the centralized orchestration will reach its limit once the number of fog nodes further increases.

4.2 VirIoT ThingVisor Factory

This section introduces the design of the VirIoT ThingVisor Factory which, in addition to FogFlow-based ThingVisor Factory, also helps implementing complex ThingVisors. The VirIoT ThingVisor Factory adopts service function chaining-based ThingVisor construction, or *ThingVisor Chaining*, where a chain of ThingVisors is used to implement a set of vThings, some of which are output of the intermediate ThingVisors in the chain. The VirIoT ThingVisor Factory supports both Information Centric Networking (ICN) and

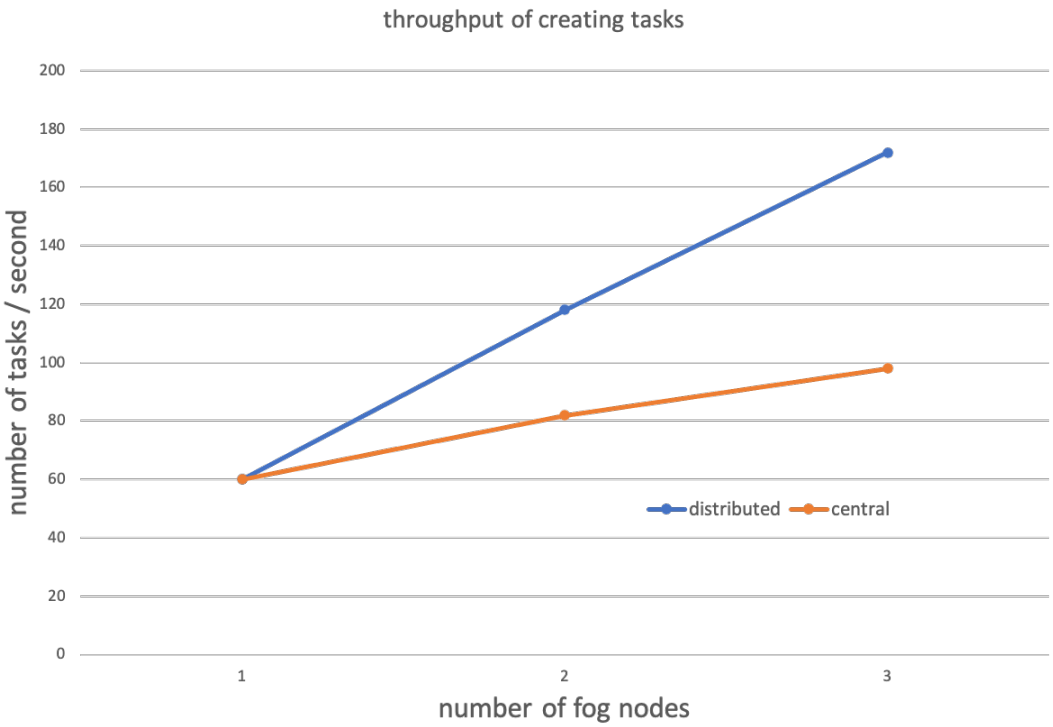


Figure 47: Speed of orchestrating new tasks on the fly in FogFlow

publish/subscribe, as possible communication model of its underlying networking technology. ICN is best used to describe ThingVisor chains which would cause large amount of topic definitions if implemented with the publish/subscribe model. In addition, ICN supports content caching. Exploiting the caching capability has potential to reduce latency, as well as network and computing load. Some results of cache exploitation are also presented in this section.

4.2.1 ThingVisor Factory Overview

Figure 48 illustrates the architecture of the VirIoT ThingVisor Factory. As shown in the figure, VirIoT ThingVisor Factory is composed by four function blocks: Service Designer, Service Deploy Manager, Service Image Factory, and ThingVisor Factory Controller. Although we have already reported the architecture in deliverable D2.3, in this section we give more detailed explanations, especially in terms of how we design ThingVisor chains, and we illustrate deployment procedures.

4.2.1.1 Service Designer

Service Designer is the most important component when designing the ThingVisor chaining. Services are designed as a ThingVisor chain, by following an analogy of flow-based programming, which is also known as visual programming. In the flow-based programming framework, such as NodeRED, a user can write a program by dragging and dropping (pre-defined) function blocks and connecting them by lines (to represent the dataflow). Note that the user can also define his/her own function blocks. By following this concept,

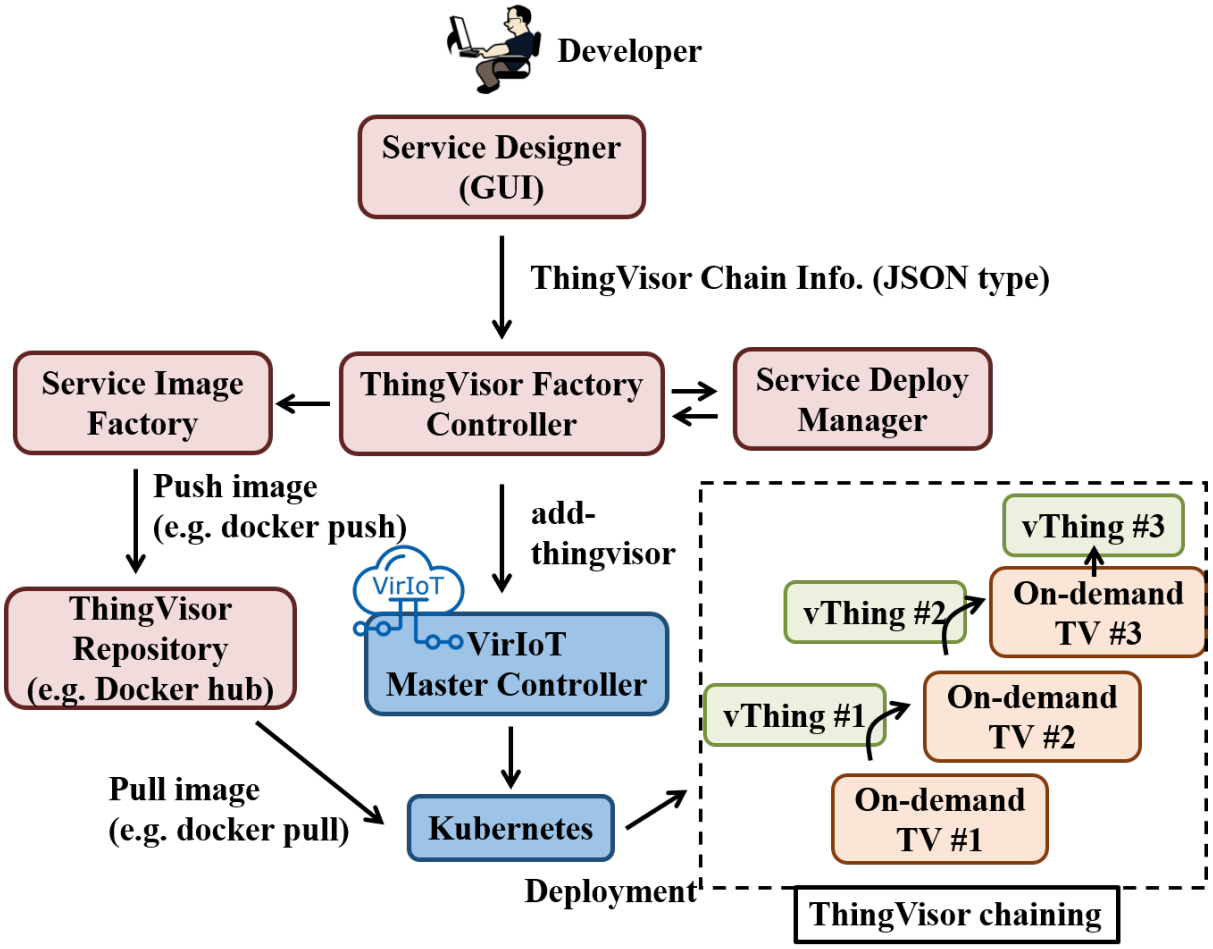


Figure 48: Architecture of the VirIoT ThingVisor Factory

as shown in Figure 49, the Service Designer has a graphical user interface (GUI) running on the web browser. In the Service Designer, a developer can develop his/her ThingVisor by using four types of function blocks: sensor, service function, connector, and program.

- Sensor blocks indicate sensor devices, and they are capable of collecting sensor data. Sensor blocks also wrap vThings: the sensor block can use the `/addVThing` API to attach to a vThing by subscribing to VirIoT's MQTT data distribution system and obtain vThing's data streams.
- Service function blocks indicate data processing and image processing. The block can also be a ThingVisor itself.
- Connector blocks represent network protocols. They are used for determining network connections between ThingVisors (or service function blocks). Currently, Service Designer offers HTTP for REST, MQTT and Apache kafka for Pub/Sub, and Cefore and NDN for Information Centric Networking.
- Program blocks are templates for Python program codes. They are used for developing ThingVisors (or service functions) from scratch.

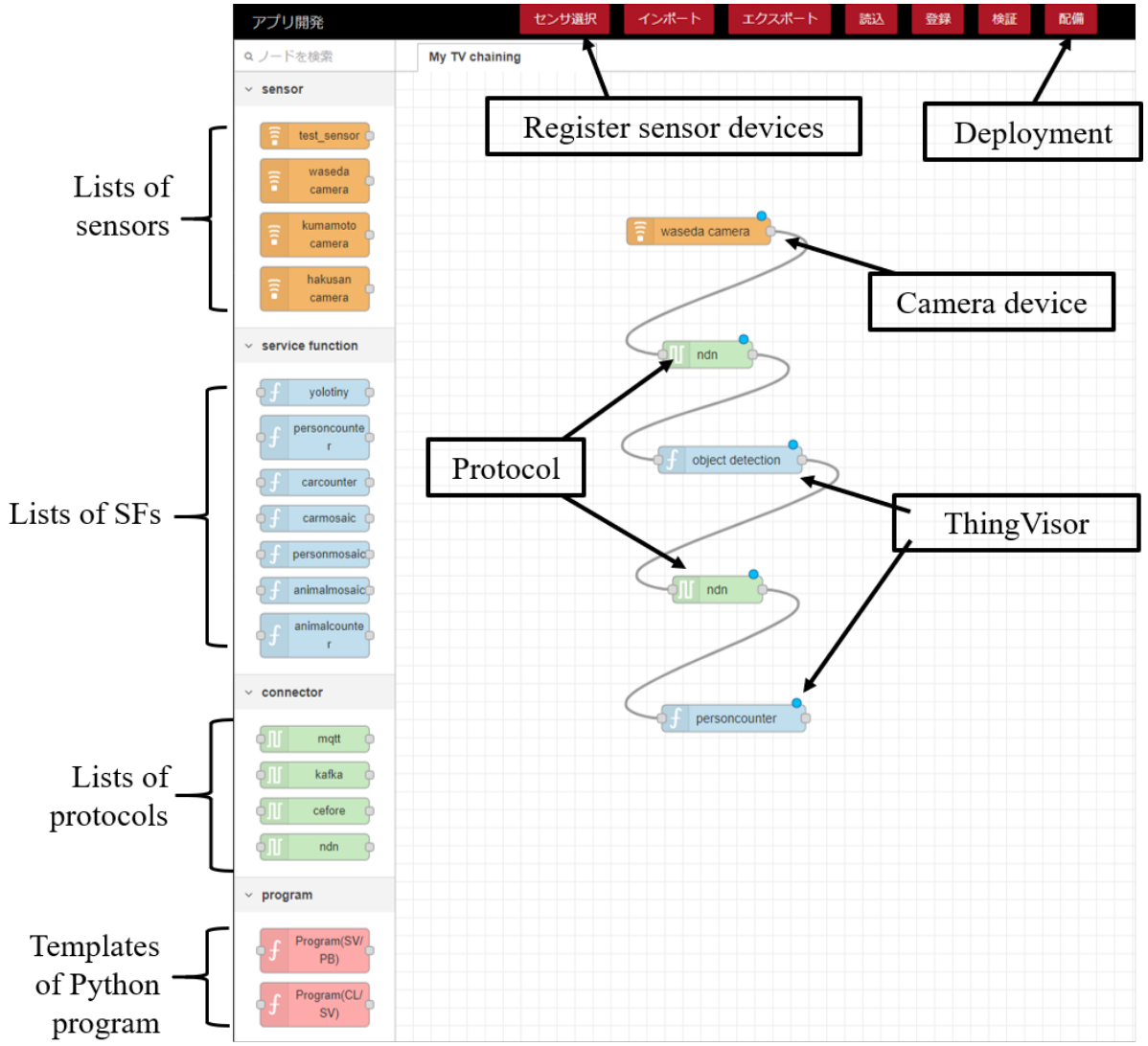


Figure 49: GUI for Service Designer

Figure 49 demonstrates an example of ThingVisor chaining by using the Service Designer. As shown in the figure, we construct a "Person Counter ThingVisor", which can count the current number of persons detected on a surveillance camera video frames. The first function block is named "waseda camera" and it denotes the camera placed at Waseda University. The captured image will be input to the object detection function block, and the resulting processed data will be input to the person counter function block, and then the result gets published as a vThing. In this example, each block is connected by "ndn", so, each data is transferred using the ICN NDN protocol.

The above, shows that the developer flexibly designs his/her ThingVisors and publishes the data, including intermediate results, as vThings.

4.2.1.2 Deployment Procedure for ThingVisor Chaining

VirIoT ThingVisor Factory is capable of not only designing ThingVisor chaining but also deploying it to VirIoT (and native Kubernetes) environment. Figure 50 indicates

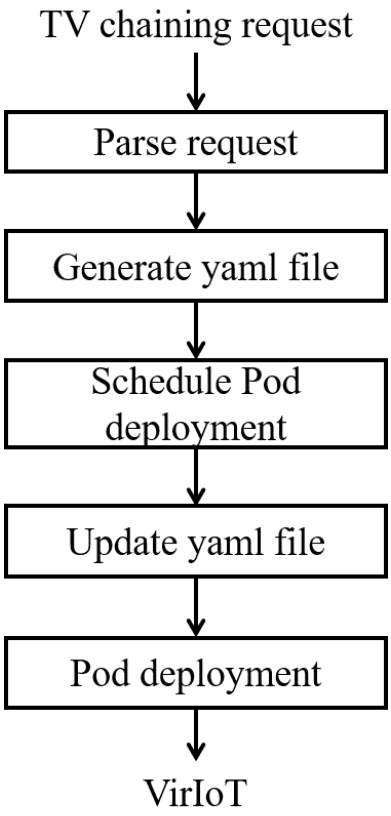


Figure 50: Deploy procedure of ThingVisor Chaining

the overall procedure of ThingVisor chaining deployment from ThingVisor Factory. As shown in the figure, there are five steps: 1) parse request, 2) generate yaml file, 3) schedule pod deployment, 4) update yaml file, and 5) deployment. All operations are defined as APIs and can be called by ThingVisor Factory Controller.

The parse request operation is to receive ThingVisor chaining request from Service Designer and transform the request in order to hand it to VirIoT Master Controller or Kubernetes master. Figure 51 shows an example of parsing of the Person Counter ThingVisor that was created in the previous section. As shown in the figure, ThingVisor chaining is represented as a sequential chain, and the controller can obtain the information of needed docker names, and parameters, such as tag name for data processing and interest (topic) name for the NDN protocol.

From this information, ThingVisor Factory Controller generates yaml files and schedules for POD deployment. It should be noted that detailed explanation of POD deployment is shown in the next section. Once all operations are done, ThingVisor Factory controller attempts to deploy ThingVisor Chaining to VirIoT, by calling the Master Controller APIs.

4.2.2 ThingVisor Chaining

Service function chaining (SFC) is a technology to compose network functions by chaining component sub-functions. With the advancement of computer hardware technology, even

Figure 51: Example of parse ThingVisor Chaining request

functions to implement networking services, such as packet forwarding, firewall, and load balancing, can be implemented by means of high-level software programming. Such technology is called software-defined networking (SDN) or network-function virtualization (NFV) depending on the focus of the topic. Several networking services are performed in sequence in data centers. For example, a packet requesting a web page is first processed by the firewall to check the sanity of the packet, and then passed to L7 load-balancer to forward the packet to an appropriate server.

The same technology can be applied to realize vThings by means of ThingVisors. A video frame captured by a surveillance camera may be processed by a ThingVisor to detect a human or an animal, and to extract a human face or an animal image, and then the output picture is further processed by another ThingVisor to identify the age of the person or to classify the animal. The information may be further forwarded to yet another ThingVisor to generate statistical data.

As stated in the paragraph above, SFC is used to construct ThingVisors (Figure 52).

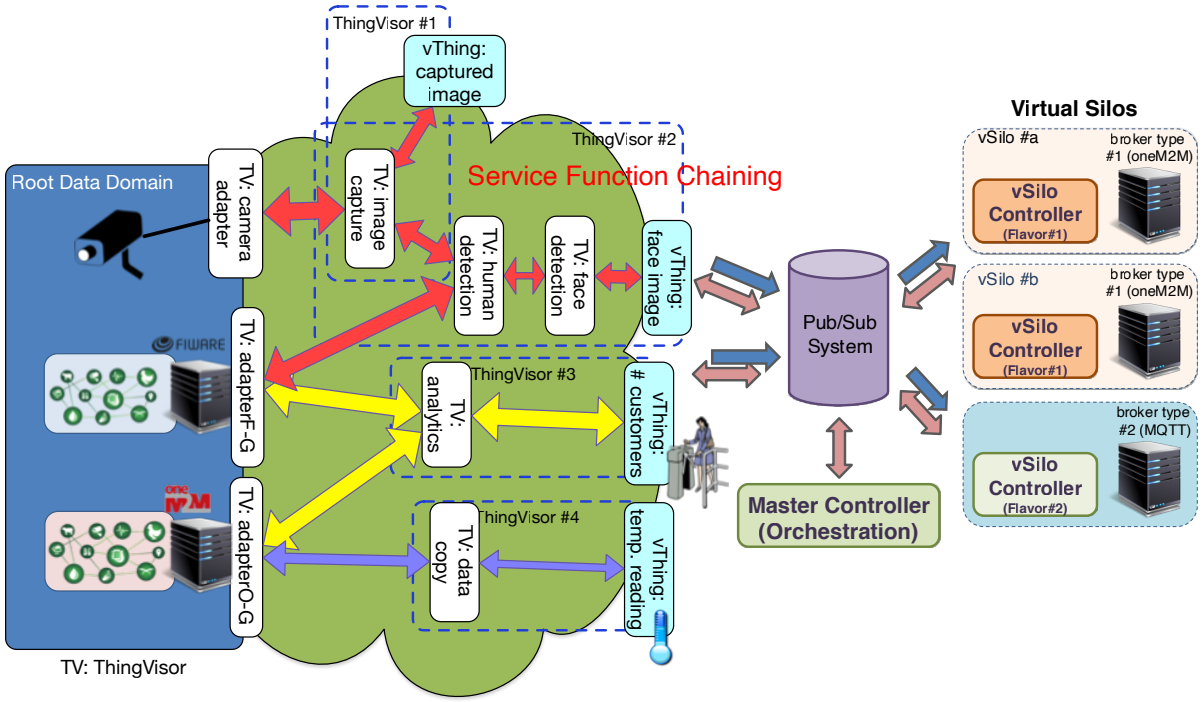


Figure 52: Service Function Chaining in VirIoT

We call applying SFC to construct ThingVisors *ThingVisor Chaining*. Video images provided by the camera directly connected to VirIoT environment are captured by the image capture ThingVisor. The captured images, as well as the images provided by one of the Root Data Domain, are processed by the human detection ThingVisor to find humans in the captured images. Then the detected human images are further processed to extract faces. The captured images by the directly connected camera are provided to Virtual Silos as vThing data. The extracted face images are also provided as a vThing to Virtual Silos. Each output of the sequence of ThingVisors executed by the ThingVisor Chaining can be provided as a vThing, too.

There are a few technical challenges in implementing ThingVisor Chaining. They are:

- the selection of the locations to execute ThingVisors and the choice of available ThingVisor instances to be used in a particular chain, and
- the communication mechanism to form ThingVisor chains.

In the following sections, we discuss each of the issues above. In addition, in-network caching in ThingVisor Chaining is discussed. In-network caching is a feature of Information Centric Networking which we adopt as one of the communication mechanism to form ThingVisor chains.

4.2.2.1 Deployment and Selection

An instance of each ThingVisor in a chain is required to be placed on a node such as a router or an edge computing node, or a VM before processing the chain of ThingVisors. If no node have any instance of a ThingVisor, it must be downloaded from a *ThingVisor*

Repository to the node. In this context, the downloading time of a large size ThingVisor can be a bottleneck of the whole processing time for the chain. In another context, if multiple chains must be processed simultaneously, several ThingVisor instances with the same type may be used by different ThingVisor chains. If such ThingVisors are processed on different nodes, ThingVisors must be downloaded to all the nodes from the repository; that is, multiple ThingVisors downloading can degrade the ThingVisor Chaining throughput. Then, one of the objectives of ThingVisor Chaining in VirIoT is to achieve effective processing for each chain, i.e., minimizing the response time with the small number of ThingVisor instances and computational resources. The points to be satisfied in VirIoT are described as follows:

- **ThingVisor deployment:**
  ThingVisor instances must be allocated to nodes that contribute to optimize the response time. This optimization becomes more crucial when no ThingVisor instance is deployed, i.e., the initial state in VirIoT. When ThingVisor instances are been deployed on a node that has already downloaded the ThingVisor, the node can initiate the ThingVisor instances without downloading the ThingVisor. ThingVisor instances should be deployed on nodes where chains can be effectively processed.

- **ThingVisor instance selection:**
  Once a ThingVisor instance is deployed on a node, it should be shared among many chains using the same ThingVisor in order to avoid redundant ThingVisor download and execution. Thus, a criterion for sharing the ThingVisor instances among chains is needed.

4.2.2.1.1 Algorithm

We designed the algorithm to perform ThingVisor deployment and ThingVisor selection simultaneously. The main idea behind the algorithm is to cluster ThingVisors as an allocation unit to a node. In particular, the ThingVisor clustering algorithm for scheduling ThingVisors, namely service function clustering for utilizing vCPUs (SF-CUV), is directed to minimize the response time with a small number of vCPUs (virtual CPUs) and ThingVisor instances. SF-CUV consists of two phases, i.e.,

1. ThingVisor clustering and pre-vCPU allocation phase (phase I).

2. ThingVisor ordering and actual ThingVisor re-allocation phase (phase II).

In phase I, an accurate scheduling priority is derived. In phase II, the accurate scheduling priority is determined and the ThingVisors are moved to another vCPU for sharing to avoid redundant ThingVisor downloading procedures. Thus, each ThingVisor can be scheduled to effectively minimize the response time with a small number of vCPUs and ThingVisor instances. In particular, the second phase minimizes the number of allocated ThingVisor instances by sharing them in the same vCPU, and it also minimizes the number of allocated vCPUs. Thus, many ThingVisor chains can be effectively and simultaneously executed in the system.

Figure 53 shows the whole procedures of SF-CUV. In this figure, the assumed ThingVisor chain is a workflow-type chain, that is a general form of SFC. Phase I outputs two

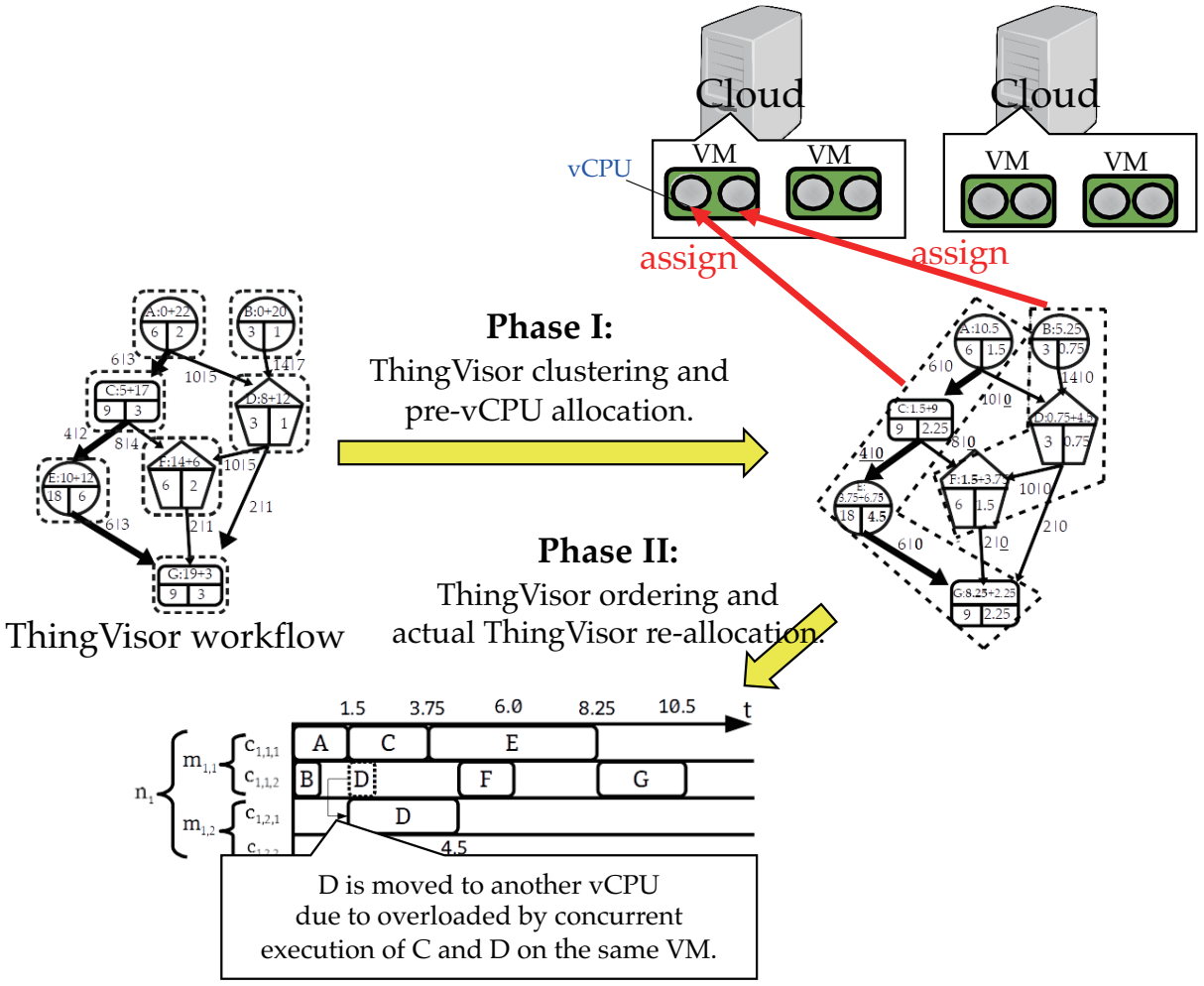
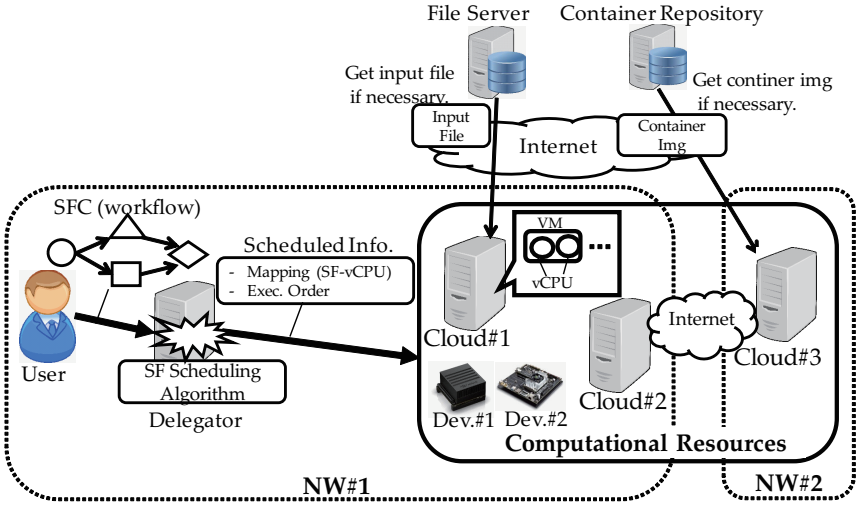


Figure 53: Procedures of SF-CUV algorithm.

ThingVisor clusters by ThingVisor clustering steps. This phase also outputs the mapping between each ThingVisor cluster and each vCPU as an allocation target. This allocation can derive an accurate scheduling priority for phase II. Then the actual ThingVisor ordering is performed at phase II. In this phase, D is moved to another VM because the concurrent execution of C and D on a VM exceeds the predefined CPU load. From those procedures, in total three vCPUs on two VMs are used for processing the SF workflow.

4.2.2.1.2 Performance Evaluation

We implemented a workflow engine to incorporate ThingVisor scheduling algorithms including SF-CUV for the performance verification in a real environment. In the following writing, service function (SF) is used to express ThingVisor since SF is the more common expression in existing research. We assume that each ThingVisor is performed in a heterogeneous distributed environment where nodes have different configurations, such as VMs in cloud, edge computing facilities, and edge devices. In this context, we conducted the performance comparisons in terms of the response time and resource utilization to verify the practicality of SF-CUV. We compared SF-CUV with five other algorithms. They are CUV-FIX, HEFT, PEFT, CAP-based, and COM-based.



Hardware	Parameter	Value
NW#1 Router#1	Bottleneck BW to external hosts	100Mbps.
	Internal BW	1Gbps
Cloud#1	# of VMs	15VMs
	# of vCPUs	2vCPUs x 4VMs 4vCPUs x 11VMs
	vCPU Freq.	{2.0, 2.5, 3.0} GHz.
	BW to Router #1	1Gbps
Cloud#2	# of VMs	6VMs
	# of vCPUs	2vCPUs x 4VMs 4vCPUs x 2VMs
	vCPU Freq.	{1.0,2.0, 2.5} GHz.
	BW to Router #1	1Gbps
Dev.#1	Type	NVIDIA Jetson AGX Xavier (2.26 GHz x 8Cores, 16GB RAM)
Dev.#2	Type	NVIDIA Jetson TX2 (2.0 GHz x 6Cores, 8GB RAM)

Hardware	Parameter	Value
NW#2 Router#2	Bottleneck BW to external hosts	100Mbps.
	Internal BW	1Gbps
Cloud#3	# of VMs	8VMs
	# of vCPUs	4vCPUs x 4VMs 4vCPUs x 2VMs
	vCPU Freq.	{1.0, 3.0, 3.5} GHz.
	BW to Router #2	1Gbps

Figure 54: Experiment Environment

CUV-FIX is a variant of SF-CUV, where each allocated vCPU in the clustering phase is fixed in the SF ordering phase. Heterogeneous earliest finish time (HEFT)[4] is a well-known task scheduling algorithm that is widely employed in real systems. Predict-EFT (PEFT)[5] is a variation of HEFT that outputs a better schedule length than HEFT. Thus, we adopted PEFT as a state-of-the art task scheduling algorithm. In capacity-based approach (CAP-based), the order for SF selection is based on the increasing order of the finish time with the average processing speed and the average communication bandwidth. Then, the selected SF is allocated to the idle time slot of the vCPU with the largest residual capacity. This approach is adopted by [6, 7]. Communication locality-based approach (COM-based) tries to minimize the communication time among SFs, i.e., the output data from one SF is sent to the nearest node having sufficient capacity to accommodate the target SF. Such data locality-based SF allocation has been reported in the literature[8, 9].

Figure 54 shows the environment in which we conducted the performance comparison. In this environment, we set up a heterogeneous computing environment in two different networks, i.e., NW♯1 and NW♯2. In these networks, we deployed computer hosts on which

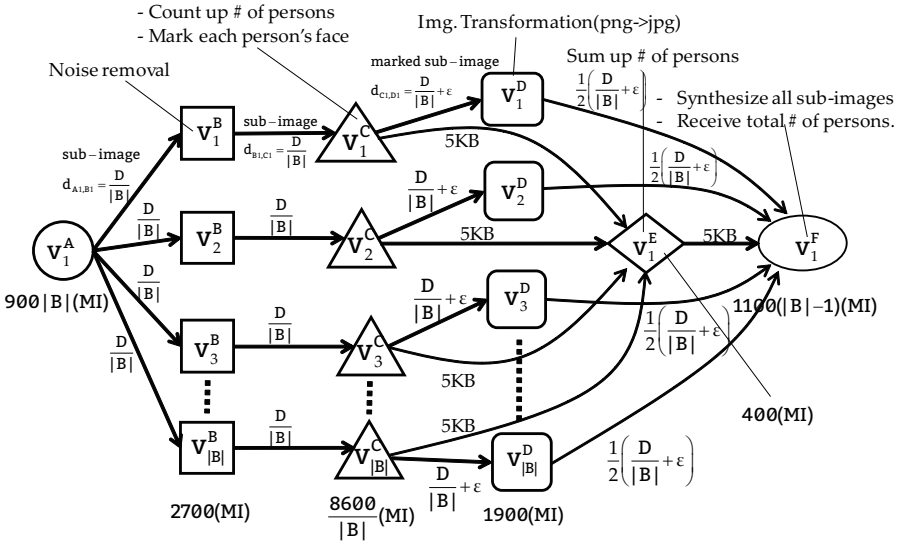


Figure 55: Applied workflow structure.

VMs run through Apache CloudStack. Each VM works on Ubuntu Desktop 18.04.3 LTS through KVM hypervisor. Although the mapping between each CPU core and each vCPU is typically controlled by a hypervisor, in this experiment, we manually mapped each CPU core and each vCPU by CPU pinning in advance. The reason is that we assume that each SF is allocated to each vCPU by an SF scheduling algorithm. In this environment, we assume that one SF corresponds to one Docker container that is stored in the "Container Repository" in Figure 54. Thus, if a node has no SF for execution, it downloads the SF from the Container Repository and then starts execution. If an execution of an SF requires one input file and the node to execute the SF has no input file, it downloads the input file from the "File Server". Thus, we assume that the file transfer involving the input file and Docker image may occur.

Figure 55 shows the applied workflow of the SFC. The workflow has six types of SF, i.e., A, B, C, D, E, and F, and every SF is executed on a Docker in which OpenCV is pre-installed. Then, we generated the OpenCV-enabled Docker image and compressed it as 1.22 GB tar file that we name "BaseSF-tar", and it was stored in the "Container Repository" in Figure 54. As the workflow has six types of SF, we generated six different Docker images based on "BaseSF-tar", i.e., every SF Docker image has the same file size (1.22 GB).

Figure 56 shows the comparison results of the degree of SF sharing in the real environment. In this figure, the horizontal axis represents the number of partitions $|B|$, i.e., the number of instances of SF B. At $|B| = 2$, the degree of SF sharing is nearly the same among the algorithms. The reason is that the number of SFs ($|V| = 3(|B| + 1) = 9$ is not sufficient to provide the difference in the degree of SF sharing, i.e., no SF has not been moved in the second phase, "SF ordering and actual vCPU allocation" phase, in SF-CUV.

If $|B|$ is larger, the difference increases and SF-CUV outperforms the others in terms of the degree of SF sharing. In particular, the difference between SF-CUV and CUV-FIX increases, i.e., more SFs have been moved to be shared in the second phase compared

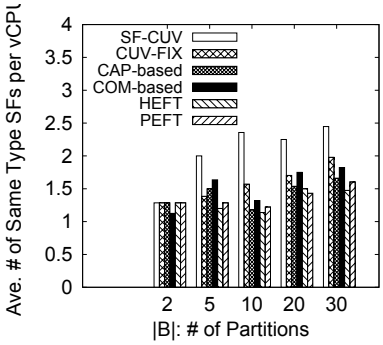
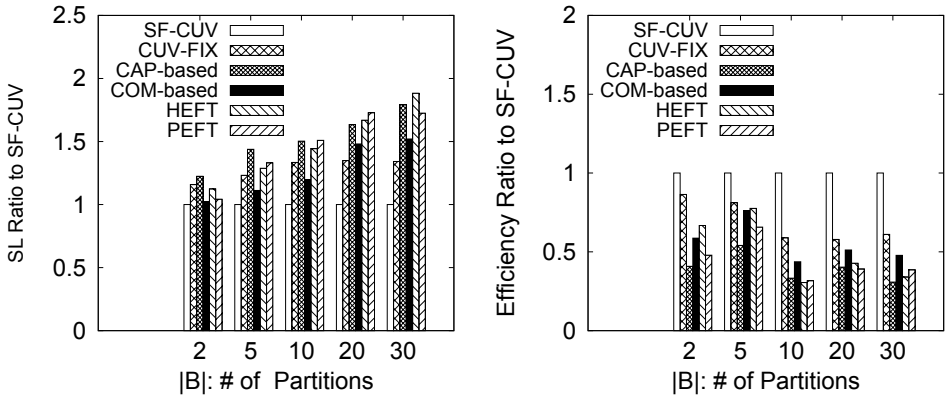


Figure 56: Degree of SF sharing.



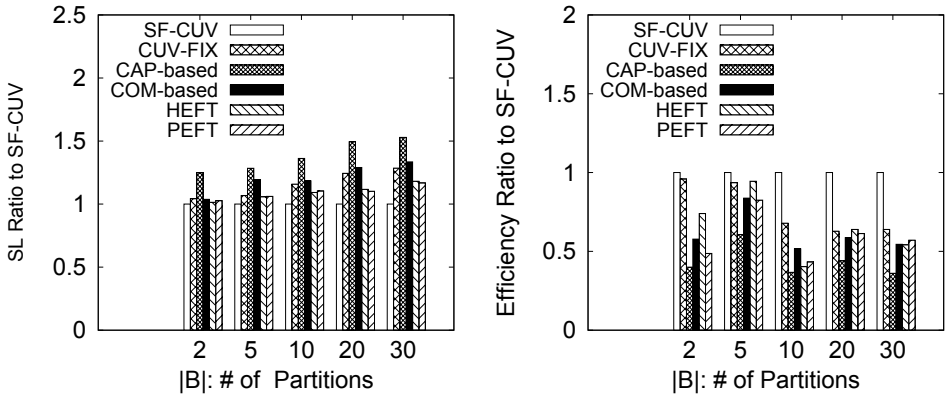
(a) response time Ratio (b) Efficiency Ratio

Figure 57: Comparisons of no SF pre-deployment.

to the case of $|B| = 2$. From this result, SF-CUV can effectively share SFs when $|B|$ is larger than 5.

Figure 57 shows the comparison results in the case of no SF pre-deployment, where Figure 57 (a) shows the comparison results for the normalized response time by setting SF-CUV response time to 1.0, and Figure 57(b) shows the results for the normalized efficiency by setting SF-CUV efficiency to 1.0. In (a) and (b), in all cases of $|B|$, SF-CUV outperforms the others in terms of the response time and efficiency. In particular, in Figure 56, the degree of SF sharing in HEFT and PEFT is worse than that in the COM-based approach, and in Figure 57(a), their response times are also worse.

Figure 58 shows the comparison results in terms of the response time and efficiency when SFs are deployed in advance. In this case, every type of SF is deployed before scheduling SFs, i.e., no SF downloading is required. Thus, this case corresponds to the ideal situation that we can obtain the information about where and which SF should be executed in advance. SF-CUV outperforms the others in Figure 58(a) and (b). From those obtained results, we conclude that SF-CUV satisfies the requirement of response time minimization with a small number of computational resources with and without SF pre-deployment.



(a) response time Ratio (b) Efficiency Ratio

Figure 58: Comparisons of SF pre-deployment.

4.2.2.2 Communications Mechanisms for ThingVisor Chaining

ThingVisor instances placed in network nodes, edge computing facilities, and clouds exchange messages to realize ThingVisor Chaining. The initiator of the message exchange in a ThingVisor chain may not be ThingVisor instances but can be external sources such as user triggers or IoT devices generating their output. The communication is not end-to-end but hop-by-hop in this respect. In this project, we are investigating three alternatives as the communication mechanism: 1) IP-based point-to-point mechanism, 2) Topic-based publish/subscribe over IP, and 3) ICN.

4.2.2.2.1 IP-based Point-to-Point

IP-based point-to-point mechanism is the simplest approach and uses IP addresses for routing of ThingVisor Chaining. To chain the ThingVisors, each ThingVisor needs to know the IP address of the next ThingVisor. In the ThingVisor workflow, we assume that a controller of the workflow knows IP addresses of every computing node in the system and assigns the appropriate IP address and port of the next ThingVisor instance after the optimal ThingVisor deployment plan is determined.

4.2.2.2.2 Topic-based Publish/subscribe over IP

Topic-based Publish/subscribe (pub/sub) is another approach to realize ThingVisor Chaining. Unlike the IP-based point-to-point, pub/sub solves the routing problem in ThingVisor Chaining by using topic name. As explained in 4.2.2.2.1, IP-based point-to-point approach assumes one controller knows IP addresses of all computing nodes, and this assumption potentially contains a scalability issue. In contrast, in the pub/sub-based approach, the routing of ThingVisor Chaining can be easily managed by only defining topics for pub/sub messages. In an experimental implementation, the topic is specified by device ids and names of ThingVisors such as "[pub/sub://]DeviceID/TV-A/TV-B/TV-C."

4.2.2.2.3 Information Centric Networking

Name or identifier assigned to a content is the address to be used to forward packets in information-centric networking (ICN). By making use of this feature of ICN, the func-

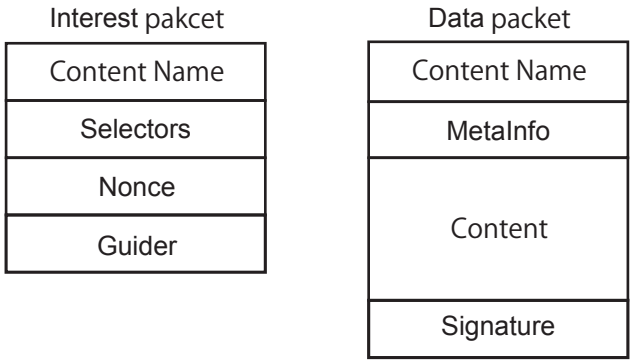


Figure 59: Original NDN Packet Format

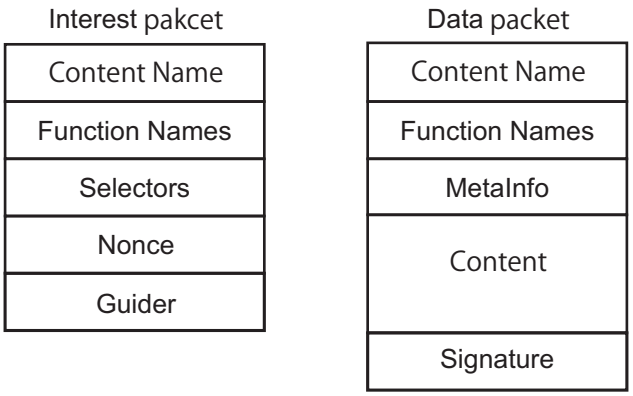


Figure 60: Extended Packet Format

tions with the same capability can be identified by the same name regardless of their implementation. IoT devices with the same functionality can also be identified by the same name. In this respect, ICN is a promising communication mechanism among IoT devices and service functions.

We adopted one typical incarnation of ICN: Named-Data Networking (NDN) or Content-Centric Networking (CCN), which uses request-response communication model. A content is requested by *Interest* packet and the content requested by the Interest packet is delivered by *Data* packet. The content to be requested is identified using a hierarchical name such as "rs1/lamp/2256" and the name is specified in both the Interest packet and Data packet. The formats of Interest packet and Data packet in NDN are shown in Figure 59.

We extended the packet format to include the names of functions, or ThingVisors, as shown in Figure 60. The Function Name field in Interest packet is used to forward the Interest packet through functions while the Function Name field in Data packet is used for caching to be explained in Section 4.2.2.3. The functions specified in the Function Names field are applied to the content requested in the Content Name field. In the context of VirIoT, both the Content Name and Function Names field can point to ThingVisor names. In the following description, the terms "function" and "service function" are used to express a ThingVisor to *process* IoT data and vThing values while "content" is used to express a ThingVisor that act as the source of data to be processed. A sequence of

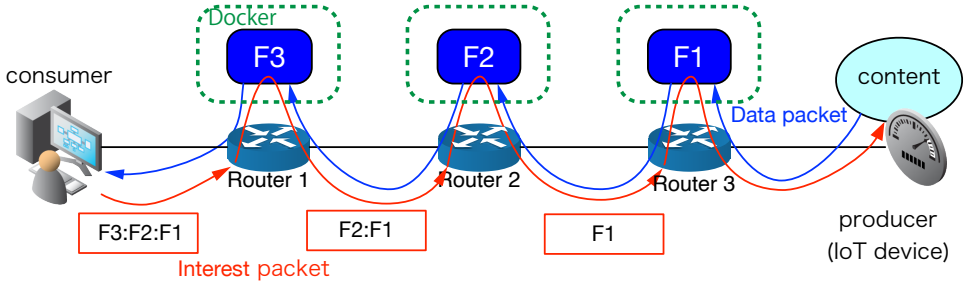


Figure 61: Example of Function Chaining

function names can be specified in the function names field. Suppose three functions F1, F2, and F3 are to be applied to the content specified in the content name field. Then, the three function names are specified in the function names field in the reverse order of the application, i.e., F3:F2:F1. The Interest packet forwarded through the three functions F3, F2, and F1 in that order, and finally to the content specified by the content name field (Figure 61). The content is first delivered to F1 in the Data packet. F1 applies its processing on the content and pass to the next function F2. F2 applies its function on the received content in the Data packet, and so on. The consumer who originally dispatched the Interest packet receives the content after application of all the three functions.

Service functions are implemented as Docker containers since they are ThingVisors in VirIoT. When a service function is installed with an NDN router, its routing table, *Forwarding Information Base* (FIB), is configured so that the Interest packet with the name of the installed service function is forwarded to the Docker container.

When the router receives an Interest packet with the name of a service function which is offered at the router, it removes the corresponding service function name from the function names field of the packet, and then forward the Interest packet to the Docker container offering the service function according to the information found in its FIB. The service function further forwards the Interest packet back to the NDN router. The NDN router then forwards the Interest packet towards the next service function by simply forwarding the packet referring to what is specified in the function names field because the next function is what is specified as the first function in the Function Names field now.

After consuming all the function names, the function names field becomes empty. Then NDN routers forward the Interest packet using its content name field. The Interest packet eventually received by the source of the content and the source creates a Data packet. The Data packet is sent back towards the consumer on the path of created by the Interest packet. Since the path includes service functions, the functions process the Data packet and further send the result back towards the consumer.

4.2.2.3 Caching in ThingVisor Chaining with ICN

One important feature of ICN is *in-network caching,* i.e., routers can temporarily memorize Data packets passing by. This feature can be exploited to improve efficiency of ThingVisor Chaining.

The output of a ThingVisor is used as an input to another ThingVisor. If a ThingVisor chain involves many ThingVisors, the production of the final output from the last

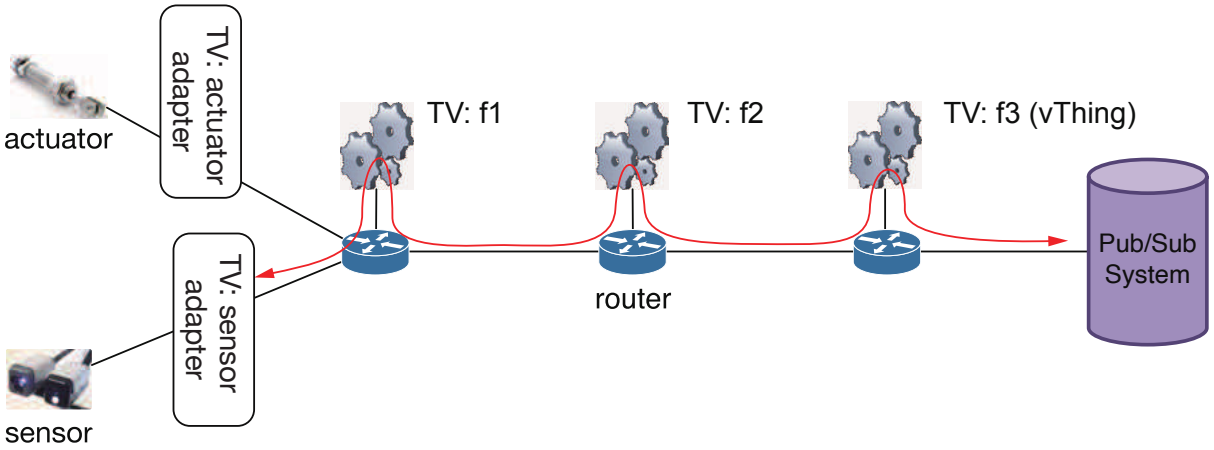


Figure 62: ThingVisor Chaining

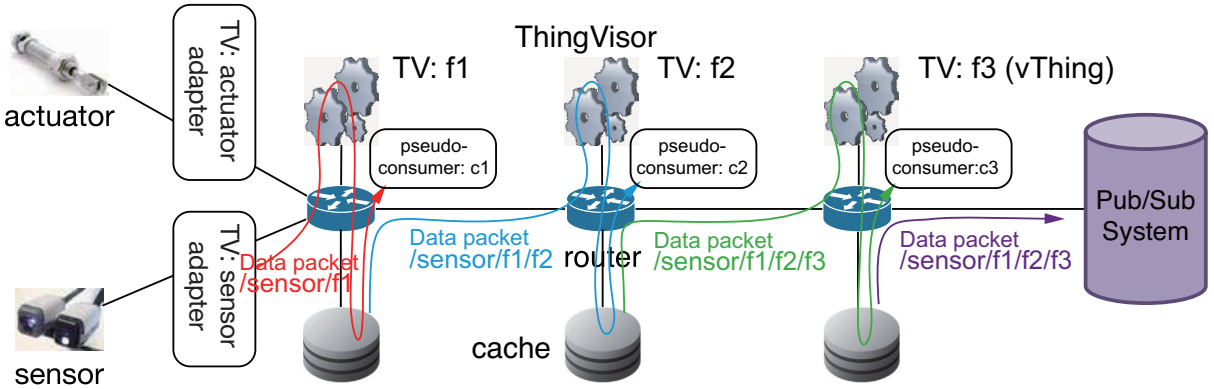


Figure 63: ThingVisor Chaining with Cache

ThingVisor in the chain takes some time after input is made to the first ThingVisor in the chain (Figure 62).

However, IoT devices are periodically activated in general. Sensors are read in a certain interval and actuators are controlled with a certain frequency depending on the applications using the devices. ThingVisor chains are performed periodically as the consequence. Each ThingVisor composing the chain is executed periodically as well.

By caching the output of a ThingVisor at the ICN router where the ThingVisor is executed, the chain can be broken into pieces (Figure 63). By executing each ThingVisor individually and periodically with proper frequency and caching the result, executions of ThingVisors are connected indirectly by cache and form a chain.

By disintegrating ThingVisor chains exploiting caching capability of ICN, output of some of the ThingVisors can be shared among multiple chains if an application of the same ThingVisor on the same data exists in multiple chains. Redundant executions of ThingVisors can be eliminated in this way while the same output from chains is maintained.

Also by the chain disintegration, the last ThingVisors in the chains can deliver the values of the vThings with minimum delay. Although this chain disintegration does not affect to the current VirIoT too much since the system assumes that IoT device readings are periodically pushed to vSilos, this minimum delay feature of cache-enabled ThingVisor

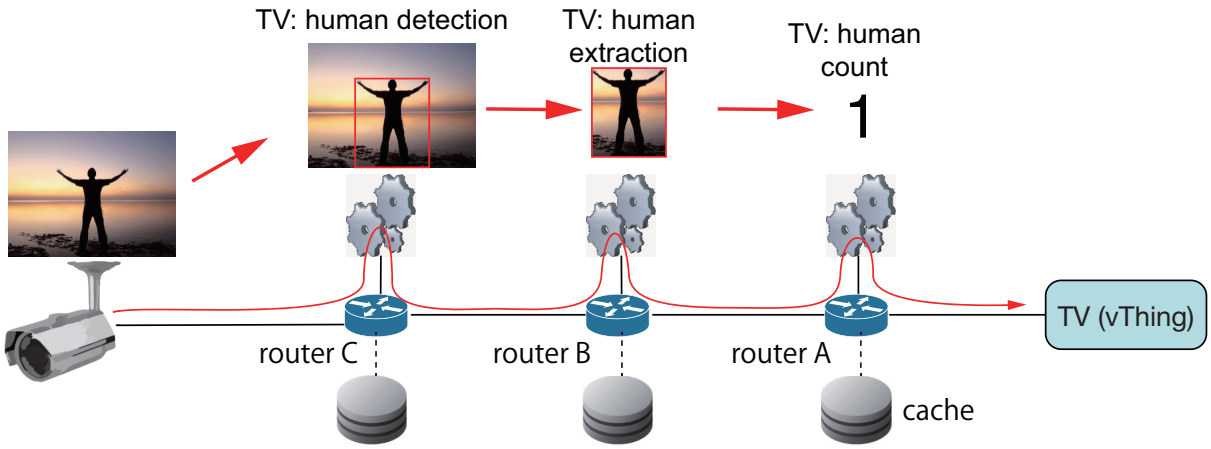


Figure 64: Experiment Configuration

chains is valuable if vThings are retrieved on demand.

A simple experiment was conducted to show the effectiveness of caching by comparing the cases with and without cache. The configuration of the experiment is shown in Figure 64. We have created five virtual machines with VirtualBox. Each virtual machine acts as a camera, ThingVisor to detect human, ThingVisor to extract human, ThingVisor to count human, and the ThingVisor to produce a vThing. Each machine also act as an ICN router.

"TV: human detection" detects all humans from the image data received from the camera, and outputs its position information in the image data and the encoded information of the image in text format. We have created "TV: human detection" based on YOLO, one of the real-time object detection software.

"TV: human extraction" decodes the image data in text format received from "TV: human detection", and cuts out only the part of the image data showing the person based on the position information of the person. Then, "TV: human extraction" encodes the cut image data into again text format, and passes to "TV: human count". "TV: human count" calculates the number of cut images (humans) based on the received information in text format, and sends that number of humans and image encoded into text format to the receiver.

We measured the time required for the "TV (vThing)" to retrieve contents with and without cache. The time the "TV (vThing)" sends the Interest packet is set to time 0. Time is counted until the "TV (vThing)" receives the Data packet. The following cases are compared.

1. All the caches are disabled.
2. Only the cache of router C is enabled.
3. Only the cache of router B is enabled.
4. Only the cache of router A is enabled.

In all cases, the content to be transferred is unified with the image data of 18.140 kB, which is the image of only one person, before applying ThingVisors. Also, before starting

the measurement, one request on the content with execution of all the ThingVisors is performed to populate the caches. In each of the above four cases, 100 measurement was performed and the average was taken. The result is shown in Table 9.

Table 9: Content Acquisition Time

case	time(ms)
without cache	112242.74
router C cache	970.09
router B cache	241.87
router A cache	81.14

According to Table 9, content acquisition time is shorter when cache is enabled. In particular, the time is greatly shortened between the case where the cache is not used and the case where the cache of the router C is enabled. This is because the execution time of "TV: human detection" is longer than other ThingVisors, and the execution time of "TV: human detection" can be saved by the cache. Likewise, when it takes time to execute a ThingVisor, by caching the Data packet after application of the ThingVisor, the time is greatly improved. Of course, the content acquisition time for the case where Data packets after application of all the ThingVisors are cached is the shortest, but even when the intermediate results are cached the content acquisition time can be greatly shortened.

Another reason the content acquisition time is shortened is that the content can be obtained from the node closer to the consumer and some network delay can be eliminated. As a consequence, network traffic can be reduced.

5 Flexible Compute Virtualization Architecture

In order to enable IoT virtualization, its applications, and orchestration, it is essential that the microservice-based Fed4IoT IoT Virtualization Platform (VirIoT) can make use of heterogeneous compute virtualization infrastructures, at scale, to deploy its services (i.e. components), beyond boundaries of domains, organizations, policies and stakeholders. We named this computing infrastructure as *Flexible Compute Virtualization (FCV)* architecture (see Figure 2), whose services should be able to i) handle heterogeneous virtualization technologies at the container level, ii) provide means for the building of different virtualization images (VM, containers, etc.), and iii) manage the orchestration of distributed computing resources formed by edge and central data centers. Towards these ends, the Flexible Compute Virtualization architecture needs to address the following four problems:

- **Resource efficiency.**
  Local, individually owned IoT infrastructures, such as those in public areas of the smart cities, offices and ordinary homes, normally have relatively small amounts of resources in comparison to centralized data centers Typically they deploy IoT gateway devices based on embedded boards, which manage sensors, cameras and/or actuation devices. Similarly, at the *edge* of big clouds, relatively a small numbers of racks covers the cloud owner's locally offered services, and they connect to the centralized data centers as much as possible. The individual servers in the racks will have high compute capacity, but at a limited scale due to physical space constraints, and yet they are required to serve millions of clients at high request rates.

- **Deployment flexibility.**
  Ideally, app developers would like to be able to deploy their new services in any platform, but in reality individual IoT infrastructures will provide different platforms: for instance, one could be based on container instances, for efficiency and manageability, and another one could be based on VM instances, for maximum isolation. It is clearly painful if the users or developers need to adapt or re-implement their services to the different platforms. In other words, we need to decouple the platform adaptation from service implementation and provisioning. But today, there exists no such framework.

- **Resource isolation.**
  IoT gateway devices and edge servers will need to accommodate compute instances, such as VMs or containers, which have different requirements in terms of performance, policies and security, within the same bare-bones hardware or cluster. Naïve design would always instantiate compute resources as VMs, which have strong isolation property between each other, or against the underlying hypervisor. However, as their footprint is very large, this strategy could result in low efficiency. On the other hand, instantiating all of the compute resources as containers, which share the OS kernel components, results in poor resource isolation in terms of performance and privacy, or violates security requirements.

- **Functional flexibility.**
  Today's applications, even those inside IoT gateways, are highly complex. For ex-

.

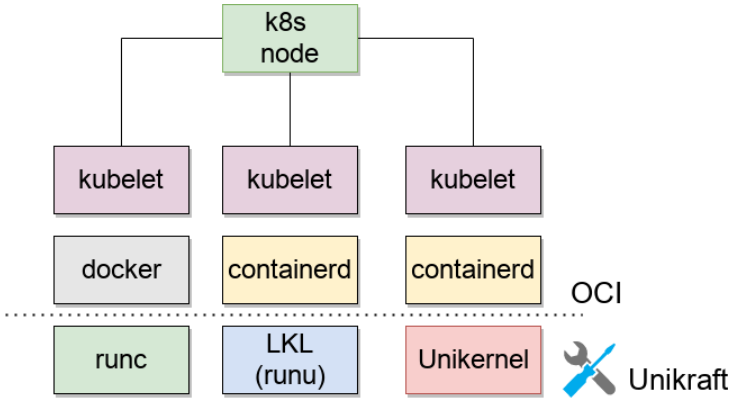


Figure 65: Kubernetes heterogeneous container runtimes

ample, the use of NGSI/oneM2M/NGSI-LD for Virtual Silos, in reality, requires the IP/TCP/TLS/HTTP protocol stack. Individual protocol implementations would also be very complex. For example, we need sophisticated congestion control and loss recovery algorithms of TCP for timely data delivery over the Internet, which is much more than the legacy TCP specification or RFC793. Further, if the devices are exposed to the Internet, they also require general security mechanisms like a firewall. At the other end of the spectrum, if the device requires extremely simple a service, for instance sending static IPv6 packets at a constant interval without delivery guarantee, the fully-fledged protocol stack is overkill. Provisioning functionality, kernel modules or libraries bloat instance footprints. This exacerbates service density and resiliency (e.g., to deploy or migrate services) issues.

In the next subsections we describe the final architecture we devised that takes into account the above problems. We start by describing the architecture and then we go deeper in two technologies, i.e. Unikernels and LibOS, that are introduced with the aim of distributing VirIoT services on low-power servers, located on the edge of the network, i.e. where the use of simple VMs or Docker containers might be impossible.

5.1 Description of the architecture

The Flexible Compute Virtualization architecture is a single distributed Kubernetes (K8S) cluster. It is flexible in the sense that heterogeneous container runtimes can be used to best fit locally available computing resources, ranging from powerful virtual machines, to low compute power devices such as Raspberry-PI boards. Figure 65 shows this concept. The K8S kubelet module that is responsible for managing containers running on a node can use several OCI-compliant runtimes: the traditional Docker runtime, i.e. runc; or other runtimes that we have developed to support Fed4IoT-specific technologies, i.e. Linux Kernel Libraries (LKL), described in Section 5.4 and Unikernel, supported by development tools such as Unikraft and described in Section 5.3.

Figure 66 shows the FCV architecture we designed to support a VirIoT platform spanning geographically distributed VirIoT *zones*. A zone is a data center or edge node where VirIoT services (K8S PODs), such as Virtual Silos or ThingVisors, can be run.

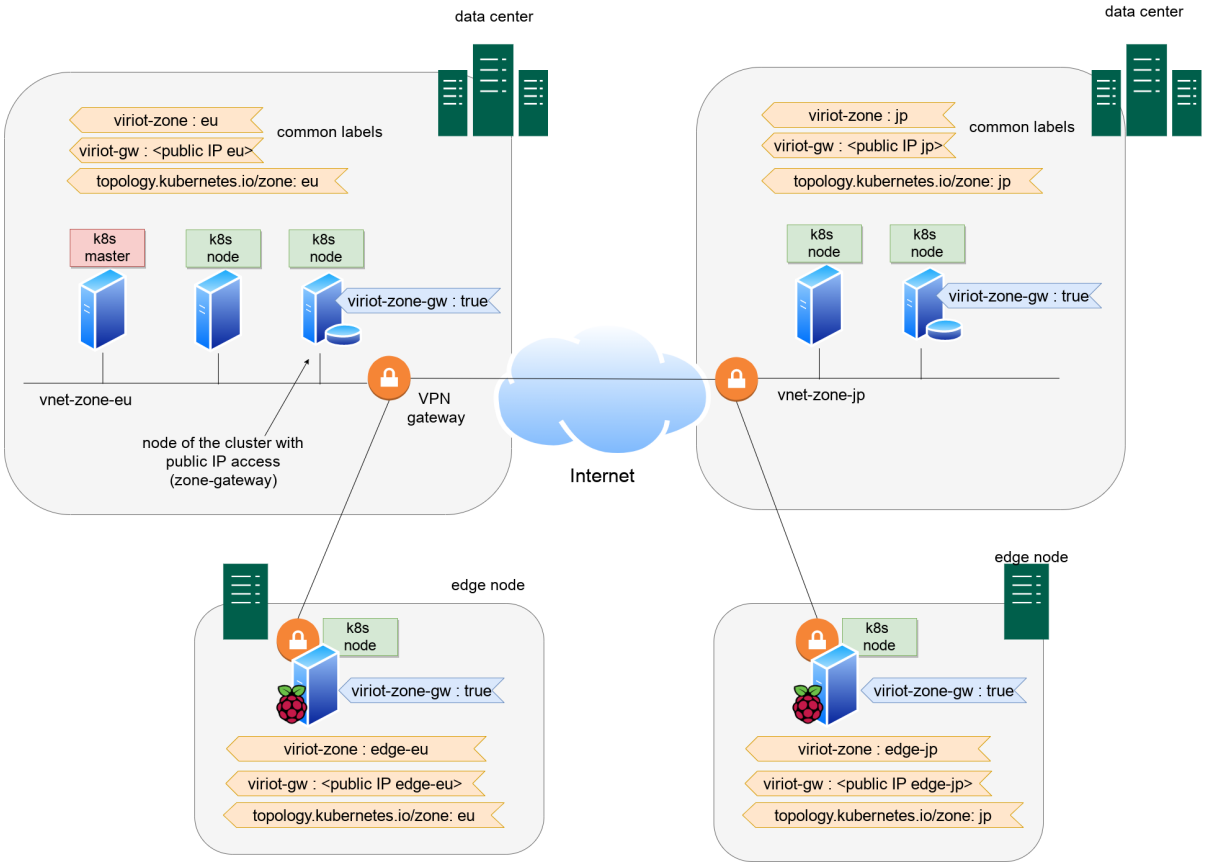


Figure 66: Flexible Compute Virtualization architecture, platform level

Zones are connected to each other through a Virtual Private Network and each zone uses a unique private addressing scheme. The IP Routing is configured so that private addresses can be used for any node-to-node communication.

Each zone has one or more Kubernetes worker nodes and there is a single Kubernetes master node. In the figure we have four zones. Two of them are data centers (e.g. one in Europe and one in Japan) and the others are on-premise standalone nodes (e.g. raspberry-PI boards) located at the edge of the network and connected to a reference data center. So we envision a two-tier architecture consisting of data centers and edge nodes. The first tier is made of data centers, connected to each other by a full mesh VPN. The second layer is made of edge nodes[5].

To support *edge computing* services and to implement MQTT/HTTP deployment systems, we devised and used specific node labels as follows.

Edge computing

VirIoT supports edge computing in the sense that it is possible to control in which zone to deploy ThingVisors and vSilos. In this way, tenants can deploy their vSilos in a specific data center closer to the end applications, with obvious benefit in terms of latency and

[5]Edge node is a generic term we use to indicate an edge zone that, however, can be composed of one or more K8S worker nodes, each connected to the VPN gateway of a reference data center.

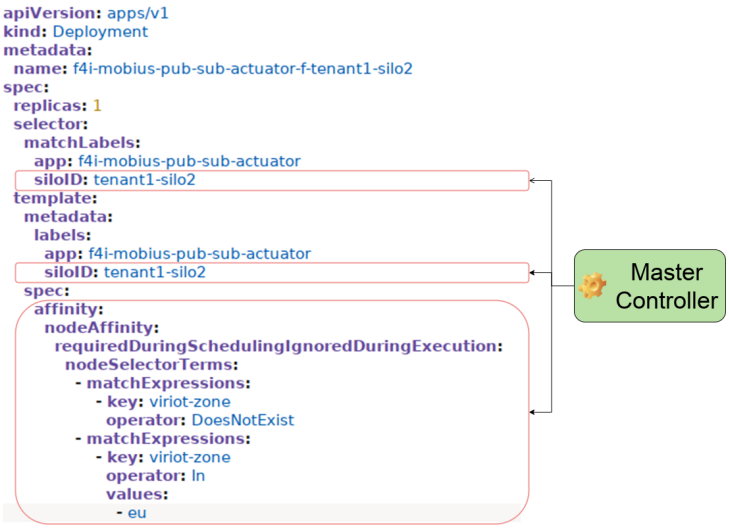


Figure 67: Examples of a part of a Kubernetes vSilo-specific YAML file of a oneM2M vSilo

bandwidth consumption. For the same reason, the administrator can deploy ThingVisors close to the real things that interact with them. Each node in a VirIoT zone has a label `viriot-zone` whose value is the zone name to be used by the API/CLI to identify the zone. In addition, each node in a zone has the label `viriot-gw` whose value is the public IP address (e.g., a cloud floating IP address) of a node in the zone from which the Kubernetes cluster can be accessed. When a user (tenant/administrator) requests to deploy a ThingVisor or Virtual Silo to a specific zone, e.g. `eu`, the Master Controller modifies the corresponding YAML file and inserts a `nodeAffinity` constraint that forces deployment on nodes labeled with `viriot-zone=eu`. Figure 67 shows an example of a portion of these changes made on the generic YAML file in Figure 10 by the Master Controller. This final YAML is then passed by the Master controller to the Kubernetes control-plane for allocation of related resources. In addition, using the K8S information and the label `viriot-gw`, the Master Controller adds into the System DB information about the gateway IP address and ports to be used to contact the ThingVisor/vSilo. Users can get back that information using the API/CLI, e.g., the commands `list-thingvisors` or `list-vsilos` CLI.

MQTT/HTTP distribution systems

As described in D4.2, and shown in Figure 4, the MQTT/HTTP distribution systems that form the internal VirIoT information sharing, have a single MQTT broker and HTTP proxy per-zone, to which the MQTT and HTTP sessions of the zone's VirIoT services should be routed. This optimizes data latency and network traffic of both context data and generic contents, by creating a multicast distribution tree that follows the platform topology. To implement this *topology routing* we used Kubernetes' internal feature named "service-topology"[6] that requires nodes to be labeled with the label `topology.kubernetes.io/zone=eu`, where the value `eu` refers to the viriot-zone to con-

[6] `https://kubernetes.io/docs/tasks/administer-cluster/enabling-service-topology`

sider. In addition, to deploy a single HTTP proxy and MQTT broker per zone, we used the label `viriot-zone-gw=true` only on one node per zone, there where we want to deploy these services, and introduced specific nodeAffinity constraints based on this label. In the figure, only two nodes in the two data centers host the MQTT brokers and HTTP proxies of the distribution systems. Therefore, traffic to the edge nodes traverses these nodes. This offloads processing on the edge nodes, which should only run ThingVisors and vSilo. If an edge node were a small data center, then it may be convenient to deploy an MQTT broker and HTTP proxy on it, by adding the label `viriot-zone-gw=true` to it.

5.2 Heterogeneous containers

At the container level, the same VirIoT services can be easily packaged by means of different virtualization/containerization technologies to adapt to different compute infrastructures. These are: Docker; Unikernels with a traditional Operating System (OS) with system-level services in the kernel space, whose building is supported by a tool named Unikraft; Library Operating Systems that are packaged as Unikernels whose Operating System has some system-level services run as as user-space libraries (Linux Kernel Libraries -LKL), as hereafter explained.

- **Unikernel**
  In our Flexible Compute Virtualization architecture, we achieve resource efficiency by enabling *Unikernel* instances wherever possible. A Unikernel is a monolithic, single-address space kernel that additionally embraces application(s) in it, without a user-space context. It thus avoids context switches that are expensive for low-latency, high-throughput services. Further, since the kernel can be application-specific and thus minimalistic, disk or memory footprints of Unikernel can be very small. Last but not least, for the same reason, many specialization opportunities in the kernel-level services are available. Unikernels can be built by using different tools, including our Unikraft, which embeds in the Unikernel image the so called Unikraft libraries.

- **Library Operating Systems**
  Typically, system-level services, such as device I/O, file systems and network protocols, are implemented as a part of the operating system kernel, usually in monolithic form. When using the Library Operating Systems (LibOSes) approach, on the other hand, we run such OS kernel components as user-space libraries, named Linux Kernel Libraries (LKL). This provides high flexibility of deployment, because portability of the user-space code is much higher than that of the kernel code. Since LibOSes usually port a full-fledged, production-quality OS kernel code into the user-space, the applications that link LibOSes can benefit from the rich functionality.

Unikernels and LibOSes concepts provide us a significant starting point to enable the flexible virtualization platform that we need especially when a low footprint is necessary, but we must address many technical challenges. Problems in Unikernels include difficulty

to build optimized instances for different applications. The developer needs to understand OS features or libraries required by the individual application.

Further, since Unikernels often require custom components, it is sometimes impossible to meet functional requirements. For example, advanced TCP features, such as TCP BBR or Rack, are impossible to use in the Unikernels available today. LibOSes, although being able to address both of the above problems with Unikernels, can only run as user-space applications, meaning that LibOS instances cannot be Unikernels.

In summary, our FCV architecture provides:

- Possibility to assemble VirIoT services as Docker containers or Unikernels that are as feature-rich as regular OS kernels.
- Ability to implement VirIoT services exploiting a LibOS both as a Unikernel, and as a real OS kernel.
- Automatic instantiation and deployment in a distributed cloud of heterogeneous optimized images (Docker, plain VMs, Unikernels, etc.).

As shown in Figure 68, the result is that VirIoT services, such as ThingVisors or Virtual Silos, requiring different system features, can be instantiated anywhere, and they can be packaged as:

- **Linux containers**, which include the VirIoT services' software, related libraries and binaries, and possibly additional libraries (Unikraft Libraries or Linux Kernel Libraries) enabling the easy repackaging of the image in different formats, e.g. Unikernel image. The supported containers will be the plain Docker ones, but also those based on Unikraft Libraries or Linux Kernel Libraries that may require a different container runtime (e.g. runu instead of runc).
- **Light Virtual Machines based on Unikernels**, which include the software of the VirIoT service and either Unikraft or Linux Kernel Libraries. Light VMs will run on top of an off-the-shelf Hypervisor (possibly bare-metal), such as Linux KVM.
- **Plain Virtual Machines**, including a fully-fledged kernel and the VirIoT service software.

The related images (containers or VMs) can be built by using the Unikraft unified tool depicted as a vertical rectangle in the rightmost position of Figure 68.

5.3 Unikraft

Specialization is arguably the most effective way to achieve outstanding performance, whether it is for achieving high throughput in network-bound applications [10, 11, 12], making language runtime environments more efficient [13, 14, 15, 16], or providing efficient container environments [17, 18], to give some examples. Even in the hardware domain, and especially with the demise of Moore's law, manufacturers are increasingly leaning towards hardware specialization to achieve ever better performance; the machine learning field is a primary exponent of this [19, 20, 21].

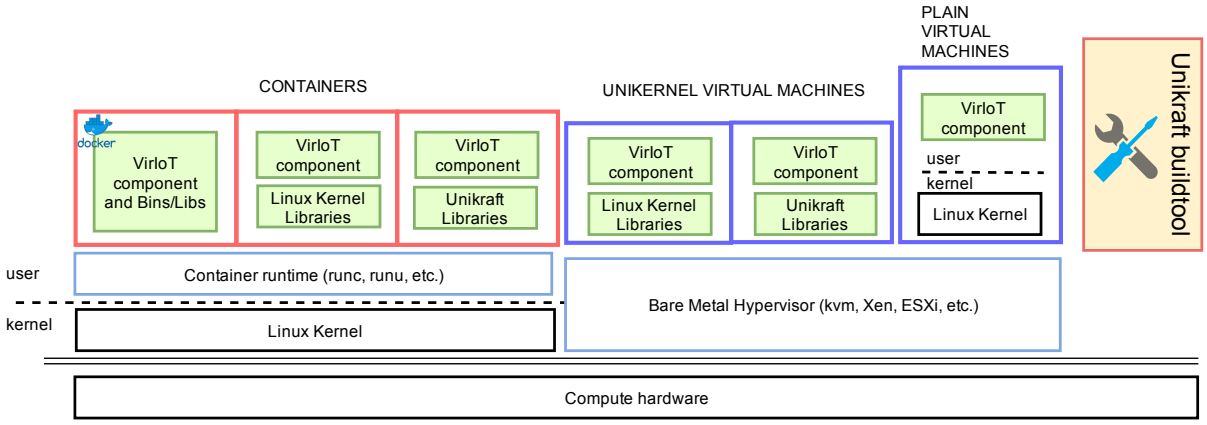


Figure 68: Flexible Compute Virtualization architecture, node level

In the virtualization domain, unikernels are the golden standard for specialization, showing impressive results in terms of throughput, memory consumption, and boot times, among others [22, 23, 24, 25, 14]. Some of those benefits come from having a single memory address space, and thus eliminating costly syscall overheads, but many of those are the result of being able to hook the application at the right level of abstraction to extract best performance: for example, a web server aiming to service millions of requests per second can access a low-level, batch-based network API rather than going with the standard but slow socket API. Such an approach has been taken in several unikernel projects but often in an ad hoc, build-and-discard manner [10, 22, 12]. In all, despite their clear benefits, unikernels suffer from two major drawbacks:

- They require significant expert work to build and to extract high performance; such work has to, for the most part, be redone for each target application.
- They are often non-POSIX compliant, requiring porting of applications and language environments.

We argue that these drawbacks are not fundamental, and we have built Unikraft, an open-source project comprising a collection of OS libraries and a build tool that enables developers to quickly create efficient unikernels for their desired applications while enabling performance comparable to, or better than, the same applications running on Linux bare-metal. Unikraft creates app+os images that are small (about 1MB) and boot quickly (1-50ms, depending on the VMM) and can run on KVM, Xen or even bare-metal.

Unikernels are small because they do not need the generality required by general-purpose, monolithic kernels that support a wide range of applications; instead, individual unikernels support only their target applications, possibly exploiting domain-specific optimization opportunities, such as known workloads. Small-footprint unikernels are particularly useful when they are deployed in resource-constrained environments, such as IoT gateways, and when massive numbers of instances (e.g., ThingVisor and VirtualSilo) must be deployed (e.g., at the IoT edge).

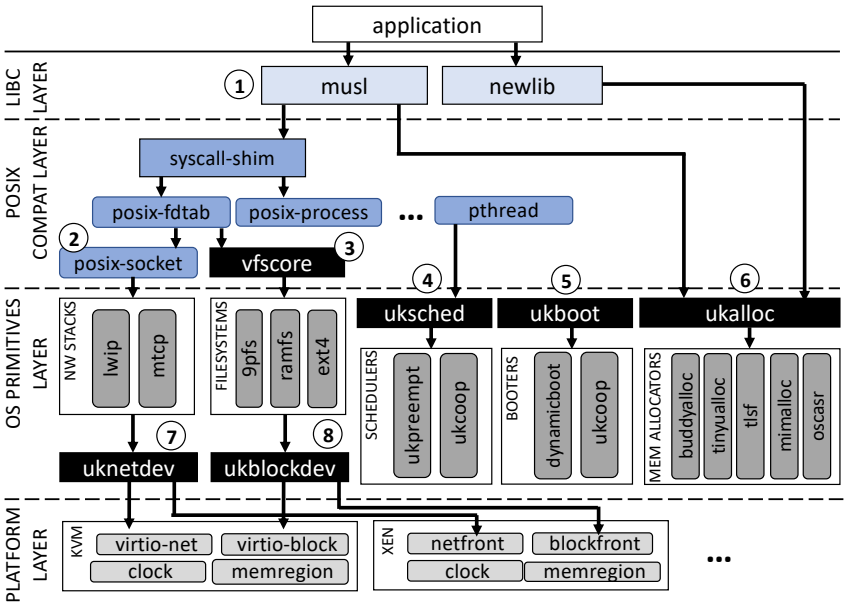


Figure 69: Unikraft architecture: all components are micro-libraries. Users select an application to build, the target platforms, the micro-libraries and Unikraft generates one image per platform. Only a subset of available micro-libraries are shown.

5.3.1 Unikraft Architecture

In contrast to classical OS work, which is split between monolithic kernels (with great performance) versus modular micro-kernels that provide great isolation between OS components (at the expense of performance), our work embraces both the monolithic design (no protection between components) and the modularity that micro-kernels advocated.

We use modularity to enable specialization, splitting OS functionality into components that only communicate across well-defined API boundaries. Our key observation is that we can obtain performance via careful API design and static linking, rather than short-circuiting API boundaries for performance. To achieve the overarching principle of modularity, Unikraft consists of two main components:

- **Micro-libraries**: Micro-libraries are software components which implement one of the core Unikraft APIs; we differentiate them from libraries in that they have minimal dependencies and can be arbitrarily small, e.g., a scheduler. All micro-libraries that implement the same API are interchangeable. One such API contains multiple memory allocators that all implement the ukalloc interface. In addition, Unikraft supports libraries that can provide functionality from external library projects (OpenSSL, musl, Protobuf [26], etc.), applications (SQLite, Redis, etc.), or even platforms (e.g., Solo5, Firecracker, Raspberry Pi 3).

- **Build system**: This provides a Kconfig-based menu for users to select which micro-libraries to use in an application build, for them to select which platform(s) and CPU architectures to target, and even configure individual micro-libs if desired. The build system then compiles all of the micro-libs, links them, and produces one binary per selected platform.

Figure 69 shows Unikraft's architecture. All components are micro-libraries that have their own Makefile and Kconfig configuration files, and so can be added to the unikernel build independently of each other[7]. APIs are also micro-libraries that can be easily enabled or disabled via a Kconfig menu; unikernels can thus compose which APIs to choose to best cater to an application's needs (e.g., an RCP-style application might turn off the `uksched` API in order to implement a high performance, run-to-completion event loop). The ability to easily swap components in and out, and to plug applications in at different levels, presents application developers with a wide range of optimization possibilities.

5.3.2 Support for Containers

Although Unikraft has a large potential, some important features required to achieve Fed4IoT's flexible compute virtualization architecture are unavailable. The first one is the support for containers managed by Docker framework. Containers are lightweight compute instances that have individual file system trees and network interfaces. Container instances share the OS kernel, but compute resources are isolated by the kernel mechanism called *cgroup*. Containers are typically managed by some frameworks, such as Docker [27] and Cloud Foundry [28], to package and deploy container images.

To integrate with the Fed4IoT virtualization stack, adding support for containers in Unikraft necessitates a top-down approach. As described in Section 4.1, Fed4IoT can use FogFlow for the design of ThingVisors, and FogFlow relies on Docker. Therefore, our goal here is that Unikraft supports Docker-based containers.

OCI Image Format.

We have decided to add support for OCI Image Format [29] (OCIIF), which is a popular container image format defined by Open Container Initiative [30], as Unikraft images. A single image consists of metadata about the contents, dependencies of the image, and filesystem changeset that describes the serialized filesystem and changes made on it.

Unikraft provides support for running a container application. Currently to run a Unikraft application within a container we need to fetch the OCI runtime, generate a runtime specific configuration file config.json and a root file system containing the application to execute. This provides us a bare minimal support for running a Unikraft application within a container. The container platform is integrated within Unikraft's build system as an external platform. The user of the platform configures the Unikraft build system to create a container image with the structure shown in Figure 70.

The build system gives user the following configuration options: With this configuration you can run a Unikraft application within a container. The Unikraft build system

[7]Unless, of course, a micro-library has a dependency on another, in which case the build system also builds the dependency.

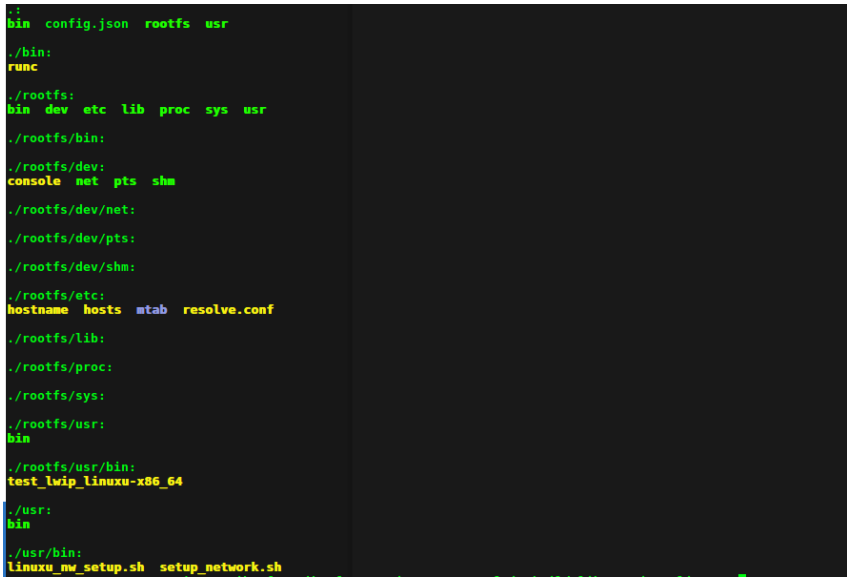


Figure 70: Unikraft Container Image Configuration.

produces the rootfs and a config.json, which the container runtime uses to run Unikraft application within a container. The rootfs of the container contains the Unikraft application and the device file needed by the application. There are no external libraries needed, the application is a self sufficient image containing all the libraries it requires. The oci runtime does not setup the network interfaces for the container application. We need to explicitly configure the network interface.

Finally, it is also worth noting that we have ongoing work to integrate Unikraft into Kubernetes so that it can be seamlessly and transparently use that framework to deploy extremely efficient Unikraft images.

Boottime networking.

Since Docker communicates with container instances for a number of reasons, we need to enable this. In particular, since the original Unikraft images are instantiated by Xen's control plane without the use of any networking features, we need to enable Unikraft images to support network interfaces and configuration.

To support network interface within a container environment we need to establish a pair of host and guest virtual Ethernet interface. The guest interface is added to the network namespace belonging the container. The guest interface acts as the network interface through which the application within the container interact with the outside world. On the host end, a bridge is created and the host end of the virtual Ethernet is configured to be a slave of the bridge device. The network is setup as a prestart hook. The hook script is called before switching the mount namespace within the container.

For Unikraft to establish a network connection in the container environment, we create a tap interface in the container environment. The tap device read onto the traffic received on the guest end of the virtual Ethernet. The tap interface is created and associate with a bridge on Unikraft boot up. Since Unikraft has the user level network stack lwip running within it, the l2 packets received on the tap device are forwarded into the lwip stack and processed by the application. The diagram below gives an overview of the design.

Figure 71: Unikraft Network Configuration.

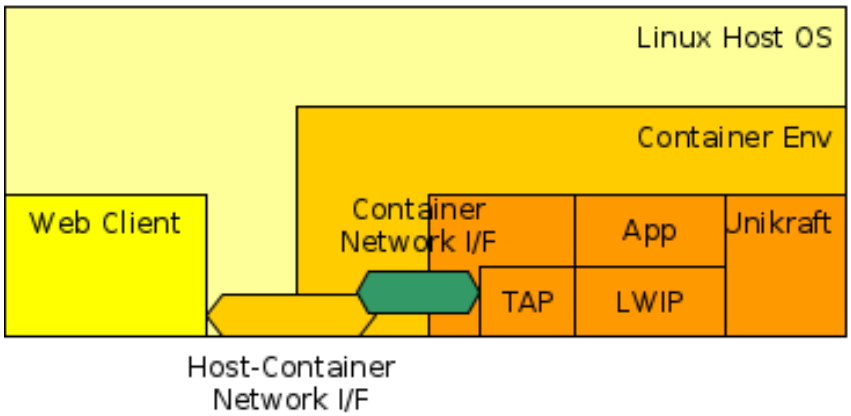


Figure 72: Unikraft Application and Networking in Container Environment.

A post start hook sets up the network interface. Figure 73 illustrates how it is established.

5.3.3 Performance Evaluation

The main goal of Unikraft is to help developers quickly and easily create resource-efficient, high-performance unikernels. In this section we evaluate to what extent Unikraft achieves this goal. Throughout our evaluation, we use an Intel Xeon E5-2690 v4 @ 2.6GHz server. All experiments were conducted by pinning a CPU core to the VM, another one to the VMM (e.g., `qemu-system`), and another one to the client tool (e.g., `wrk` or `redis-benchmark`); by disabling Hyper-threading; and by setting the governor to performance.

The results are shown in Figure 74, showing how long a helloworld unikernel needs to boot with different VMMs. Note that Unikraft's boot time (only the guest, without VMM overheads) ranges from tens to hundreds of microseconds when the VM has no devices, and up to 1ms when the VM has one networking interface. These results compare positively to previous work: MirageOS (1ms with Solo5), OSv (10ms on Firecracker with a

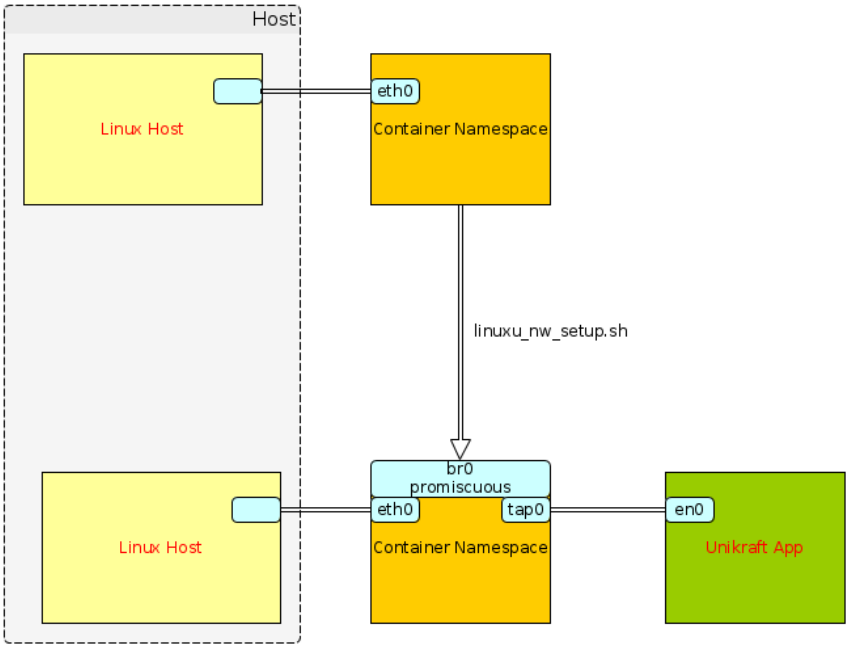


Figure 73: Host Networking Setup with Unikraft and Container.

read-only filesystem), Rump (15ms on Solo5), Hermitux (31ms on uHyve), Lupine (95ms on Firecracker, 45ms without KML), and Alpine Linux (around 200ms on Firecracker). This illustrates Unikraft's ability to only keep and initialize what is needed.

Overall, the total VM boot time is dominated by the VMM, with Firecracker and Solo5 being the fastest (1-3ms), QEMU Microvm at around 25ms and QEMU the slowest at around 65ms. We further plot Unikraft guest boot times for SQLite and Nginx, which add 1-2 ms to the total boot. These results show that Unikraft can be readily used in scenarios where just-in-time instantiation of VMs is needed.

In addition, previous work [22] stressed the importance of not only fast instantiation but also VM density. In order to understand how many unikernel VMs we could pack on a single server when RAM is a bottleneck, we ran experiments to measure the minimum amount of memory required to boot various applications as unikernels, finding that 2-6MBs of memory suffice for Unikraft guests (Figure 75).

5.4 Linux Kernel Library

5.4.1 Background

The growth of container architecture and its ecosystem, development, integration, and the deployment of programs is nowadays not a headache anymore, thanks to a powerful and flexible environment of underlying virtualization technologies. At the same time, when facing, for instance, the development of a large smart-city platform involving distributed computation across cloud and edge devices, the conventional container system meets stricter requirements of even smaller footprint of the execution environment while avoiding functional degradation to the container runtime. Although the conventional container architecture, or operating system virtualization based on software partitioning,

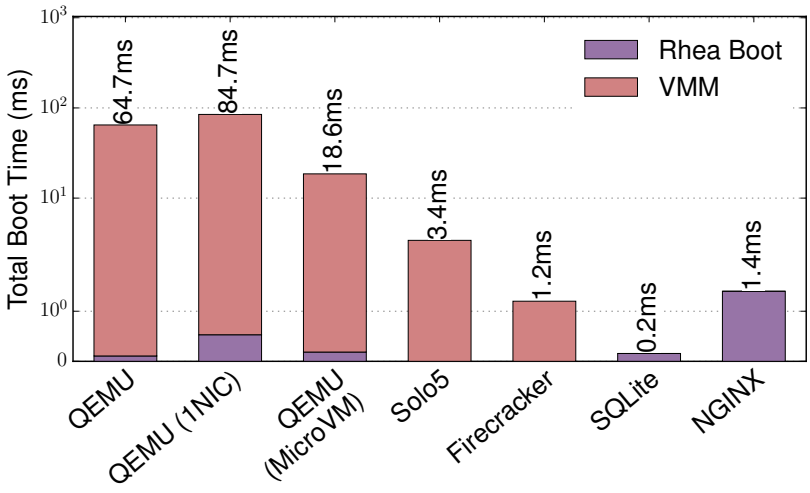


Figure 74: Boot time for Unikraft images with different virtual machine monitors.

with the namespace technique, has a lightweight advantage when compared to the full machine virtualization with hardware-assisted partitioning, containers still have room for even lighter weight when considering the execution of a set of small programs (a.k.a. micro-services), as several studies reveal [31, 32, 33, 34].

5.4.2 Existing Solutions

Prior works have also tried to address the issue. Their approaches are varied: [31] tried to reduce the size of container image by splitting into slim and fat images, where the fat image contains tools and the slim image only contains main application so that runtime footprint could be reduced. This approach could offer full feature-set of underlying host systems but supported platforms are limited to host systems if there are no emulation of the deployed images.

Unikernels [35][36][37] offer specialized, small guest kernel over hypervisor which has smaller resource usage by combining user- and kernel-space in their runtime memory layout. This integration also contributes smaller runtime overhead by eliminating context switches which happen during I/O operations. Moreover, due to its nature of hypervisor use, the restriction of underlying system is also relaxed, so that we can cover more various devices. However, even several Unikernels implementations [38][36] offer the binary compatibility to Linux, the compatibility layer is often incomplete as underlying kernel (in Unikernel) is not Linux.

According to the observation from past studies, the lightweight property of virtualization with the various platform support is achieved by some level of functional degradation. We are trying to fill this gap in our proposed software.

5.4.3 Linux Kernel Library: Rich Feature-set with Specialized Kernel

Our motivation here to develop Linux kernel fitting into our requirements to provide specialize Linux kernel feature without losing the original, mature Linux kernel. Thus

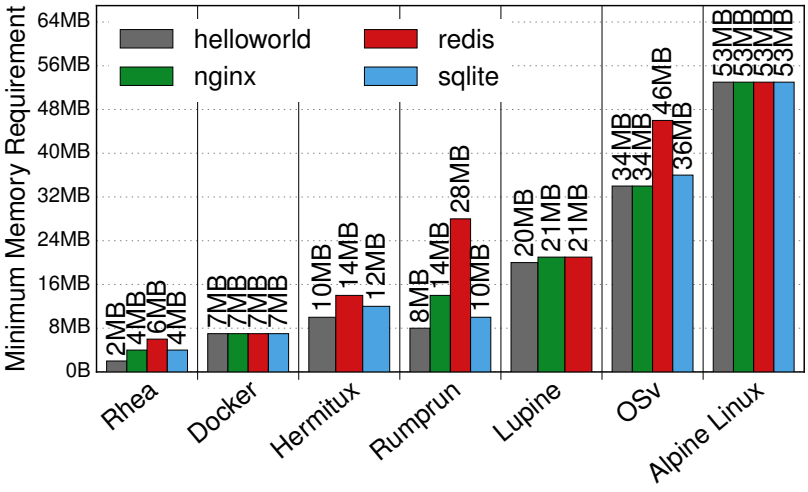


Figure 75: Minimum memory needed to run different applications using different OSes, including Unikraft.

the binary compatibility property is simply one of the feature that the original Linux kernel offers.

Our design decision follows *stand on the shoulders of giants*, where we will try to avoid writing code and rather we will try to re-shape the mature code base to address more specialized requirement. Thus, Linux Kernel Library (LKL) was designed. LKL is originally aiming to allow reusing the Linux kernel code as extensively as possible with minimal effort and reduced maintenance overhead. It was proposed around 2007 but we have recently developed the LKL so that we can address further use cases, including lightweight property of application runtime.

Unlike other userspace ports of the Linux kernel such as User-mode Linux (UML) [39], LKL aims to be a reusable library in a variety of environments so that programs can link to the components of OS features implemented in the Linux kernel as a library OS. As illustrated in Figure 76, LKL introduces a hardware-independent architecture in the Linux kernel tree by decoupling the LKL host environment (machine/environment-dependent code) from the Linux kernel, largely contributing to the platform independence of this library.

As LKL takes the minimum-additional-code approach, it has other benefits. An application running with the LKL has its richness of Linux features, such as the latest protocols or algorithms inherited from the Linux kernel. This is not possible for other supersized kernel approach as those implement Linux feature from scratch. Additionally, thanks to the decoupled design of machine/environmental dependent code into the host environment, the porting effort to non-Linux platform is relatively easy. The latest LKL can run on top of not only Linux host, but also Windows, FreeBSD, macOS, inside UEFI bootloader, Android phone (which is hard to upgrade kernel).

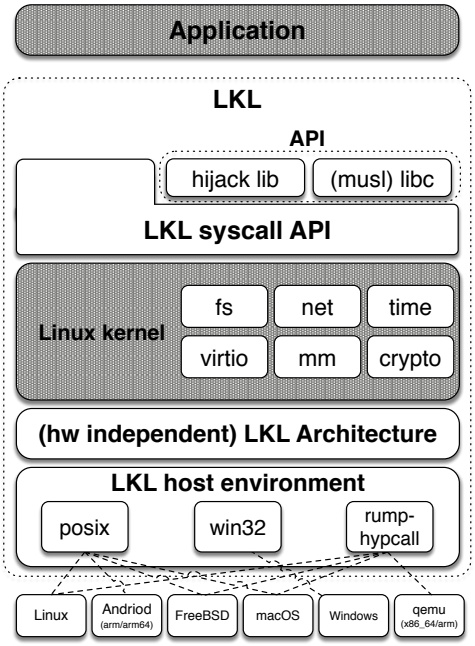


Figure 76: Structure of LKL as a portable and reusable library of the Linux kernel.

5.4.4 Container Integration

Interfacing to container infrastructure with LKL is important since VirIoT platform is already developed on Docker/Kubernetes infrastructure, thus LKL with container interface would not request changes to the VirIoT orchestrator (the Master Controller). Our docker integration is not just providing a wrapper script for LKL-ed programs, in order to use the docker client command, but is a solution more tightly coupled with the infrastructure, by offering plug-in module container runtime. With this runtime, named `runu`, users benefits of LKL with fully-featured, and customized Linux kernel for the container instance.

The implementation of the `runu` runtime has two aims: 1) to invoke a LKL-ed process that includes libOS and 2) to bridge the interface of the standardized runtime specification by Open Container Initiative [30]. The implementation allows us to replace the default runtime environment (`runc`) with our custom one via a command-line option (in the case of Docker), or the custom runtimeClass annotation (in the case of K8S), to preserve the portability of the container usage model.

Because of the cross-platform support of the Go language, the implementation of `runu` is quite straightforward on both Linux and macOS over Intel (amd64) and Arm (arm32 and arm64) platforms, which are the currently tested platforms. We also have integrated 9pfs server functionality[8] to share a filesystem layout from a container image to a container instance.

5.4.5 Performance Evaluations

Our performance evaluations consist of two parts to confirm 1) the lightweight property of container instantiation, and 2) the measurement study of feature richness of network

[8] `https://github.com/docker/go-p9p`

stack.

Minding typical serverless platform use cases, where the container instances are frequently started and terminated upon the requests from users, we measured the duration of a simple Python program execution that is ready to listen to a socket. Although the result involves various preparatory elements, such as filesystem and network interface creation and several message interactions between the container engine and runtime to manage container life-cycle, we used the `docker run` command to invoke a container instance because it reflects the typical use cases of serverless deployments.

The experiments were conducted on two machines, Dell PowerEdge R330 with a 4-core 3.8 GHz Xeon E3-1200 and 64 GB memory, interconnected via an Intel X540 10 Gbps link. The machines ran the Linux 4.18.5 kernel in the Fedora release 28 and used Docker 18.06.1-ce for the container framework. To compare `runu` (our implementation for container runtime to instantiate LKL-ed applications) with other approaches, we used the Kata container [40] version 1.8.0 with the 4.19.28-48.1 Linux kernel, gVisor [41] from the git e9ea7230 revision, Nabla containers [42] from git 2cecc88 revision, a native Linux application on the host kernel without containers, and `runu` runtime environment. The base Linux kernel in the LKL was version 4.19.0.

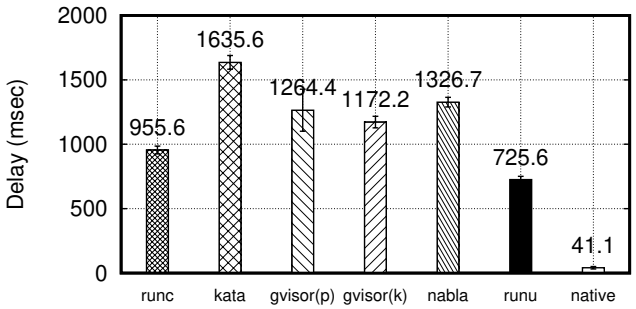


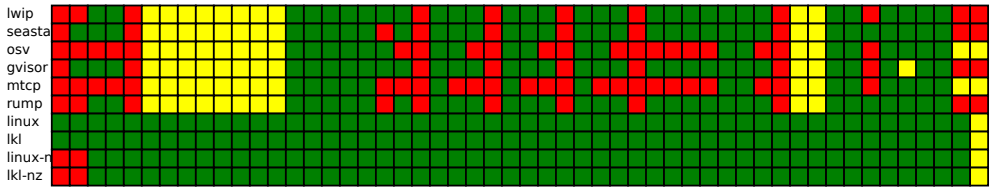
Figure 77: The duration of Python script execution from 30 measurement iterations (with the mean values).

Figure 77 plots the result of this measurement with the standard deviation from 30 repetitions. The duration of the (Linux) native Python program is about 41 ms, and this can be used as a baseline for typical process instantiation in the host system. The standard Docker runtime environment (`runc`) takes 956 ms, requiring configurations such as signal handler installations, system call filtering to the host kernel, followed by multiple process invocations. gVisor (`gvisor`(p), gVisor with ptrace system call trap, and `gvisor`(k), with a trap via a KVM) do not utilize the namespace facility, but require even more time (1264 ms and 1172 ms) because of the overhead of its context switches across multiple system calls. Nabla container (`nabla`) shows longer duration to others except Kata, 1327 ms, and this is also slower than that presented in their paper [37] because our measurement involves multiple containers and OCI runtime interactions while their paper may not include these considerations. The Kata container (`kata`) takes 1635 ms, and this is shorter than a typical virtual-machine instantiation but slightly longer than that required by the other approaches and represents the cost of transparent hardware virtualization. `runu` runtime environment takes 726 ms, and this is the fastest among all OCI runtimes including the result of `runc`, although it also involves an additional

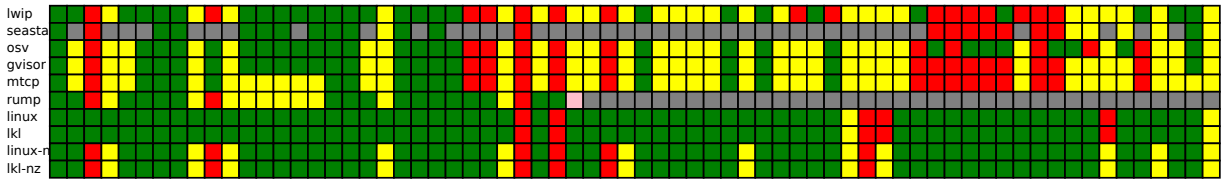
filesystem mount to load Python library files.

Next, we tried to measure the level of maturity of network stack implementation by testing the conformance of the implementation. We used Ixia IxANVL (automated network validation library) [43], a software utility, to validate network protocol compliance and interoperability. By running a set of test suites to verify the behavior of network stack, based on standard specifications of IETF RFCs, the tool reports the number of successful tests for each network stack implementation. We use the reported number as the level of maturity of network stacks.

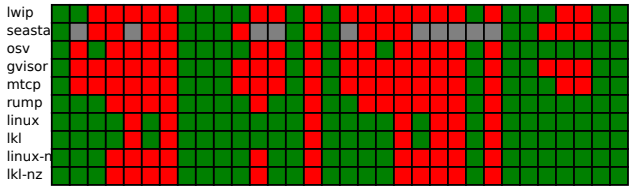
We used lwIP (git 7b7bc349 revision), Seastar (git c19219ed revision), OSv version v0.24, gVisor (git faa34a0 revision), mTCP (git 611cc05d revision), rump kernel (git f10683c revision of buildrump.sh), LKL (git 5221c547af3d revision- based on Linux 4.16.0 version), and native Linux kernel (4.15.0-34 version). We used IxANVL version 9.19.9.32 (Linux) and ran the following test suites for the conformance tests: ARP, IPv4, and ICMPv4.



(a) ARP



(b) IPv4



(c) ICMPv4

Figure 78: Conformance test results (IxANVL) for network protocol based on RFC specifications (Pass=green, Failed=red/yellow).

Figure 78 visualizes the result of this measurement. The matrix in the figure is interpreted as follows: the x-axis is a sequence of test cases, and green box indicates that a test is passed (succeed) while red and yellow box are failed tests. The gray box indicates that the test are not conducted due to the restriction of a particular implementation (e.g., following tests are not able to conduct due to previous test's failure). Of all the tests conducted, the conformance of LKL is the best to the others while achieving the identical results with the test of typical Linux kernel.

5.4.6 Resource Minimization of LKL-based container

We conducted evaluations on the resource usage with LKL. Specifically, we focus on the storage and memory footprint for this evaluation because small storage footprint will contribute the shorter download time of container images and speedup the startup time of applications, while small memory footprint contributes to relax runtime limitations under resource-constrained environments.

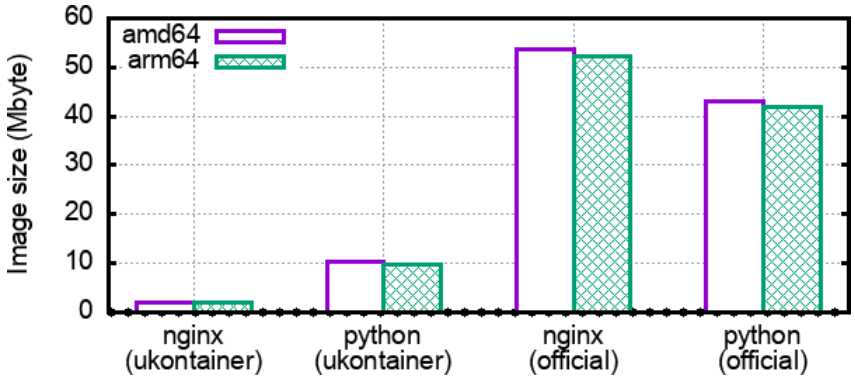


Figure 79: Size of container images: comparison between LKL-specific images (ukontainer) and official images published under Docker hub (both Amd64 and Arm64 platform).

First, we measured the image size of containers with LKL-specialized runtime. With LKL, programs can run with a single process with auxiliary files (e.g., configurations, plugins of language runtimes), thus we can shrink the image by eliminating unused files for the programs. We did two ways of analysis for the image-size reduction: static-analysis focuses on the build process of LKL and eliminates the unused functions and codes to reduce the file size of final products, while dynamic-analysis more focuses on detecting unnecessary files of language runtime, in this case python, by executing programs with runtime profiler and delete unused files reported by the runtime profiler from the container images so that the image size will be shrunk.

Figure 79 represents the result of measurement. We measure the container images with `nginx` and `python`, as two representative images with two different platforms (amd64/linux and arm64/linux). We compared LKL-specific images with nginx official image (version stable) and python official image (version 3.9.1-slim). With the nginx image, the image size of LKL-specific container is about 3.6% of the official image in both platform (amd64 and arm64). This is largely due to the characteristics of a typical container image which is based on a Linux distribution (Debian in the nginx case), resulting including files of other dependent packages. With the python image, the image size of LKL-specific container is about 23.5% of the official image. The reason of not seeing similar improvement with the nginx image is that the required files of python runtime is almost similar and both of images (LKL and the official one) include the files, resulting smaller improvement. While applying the dynamic-analysis with the python programs, the image size can be shrunk to about 4MB, but the resulting image is heavily specialized and is not reusable with different scripts, thus we did not apply this technique in this measurement.

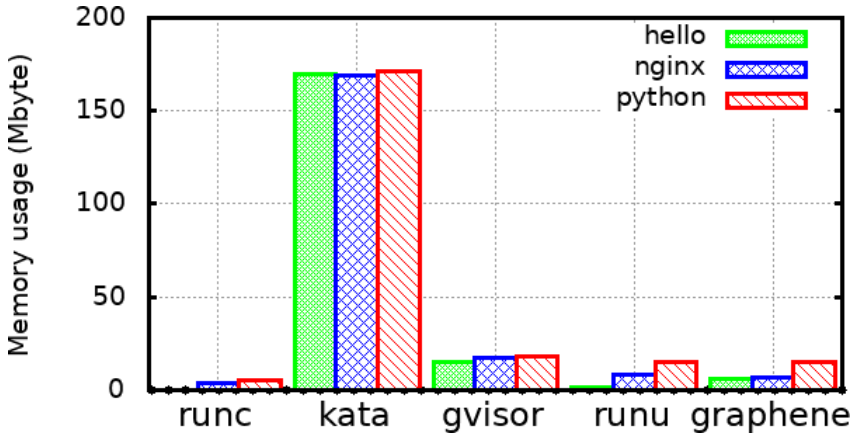


Figure 80: Memory resource footprint with different runtimes and different programs.

We also measure the memory usage of programs with different container runtimes in order to see how LKL-based container performs. We used three different programs for this measurement: a simple program to print one-line of strings and sleep for five seconds (i.e., hello), the nginx program after the initialization (nginx), and a python program with printing a single line, as hello does (python). For the runtimes, we used conventional container runtimes (runc), a runtime based on virtual machines (kata container), a kernel emulator called gVisor, a library OS, called Graphene, offering Linux ABI compatiblity (graphene), and our LKL-based runtime (runu). Figure 80 plots the result of the measurement; since we tried to understand the difference in memory consumption between container runtimes, we only measured the resident set size of the main programs to eliminate the other consumption by control processes (e.g., `containerd-shim`, OCI runtimes, and helper programs such as `runsc-gofer` in `gvisor`). As we can see, runc consumes the smallest size of memory because it only reports the memory usage of userspace programs, which does not account for the usage of kernel space resources. Other four variants contain the kernel-related usage. gVisor, Grapahene, and runu show almost comparable results while Kata container consumes much memory than the others. While kata container already applied optimizations to typical VMM for the resource consumption, it still requires a reasonable amount of memory compared to other userspace-based techniques.

5.4.7 Kubernetes Integration

Our dedicated container runtime, runu, is compliant to the OCI standard from the beginning and has been applicable to the VirIoT system, which can be deployed over Docker infrastructure. However, the runtime was not ready to deploy via Kubernetes (K8S) cluster due to additional consideration on the design of the OCI runtime implementation. We have extended the runu runtime to support container operations over K8S. The extension focuses on the auto-configuration of network information (IP addresses, routing tables), and preparation of the runtime installation to the CRI (Container Runtime Interface) runtime.

6 Conclusion

This deliverable has described the many details of the the Fed4IoT Virtualization Stack, including both IoT (giving details on top of what was previously reported in D2.3) and compute virtualization aspects. The whole system is centered around a plethora of facilities, that live inside a platform we called VirIoT. We also addressed how to ease development and deployment tasks for VirIoT, by means of FogwFlow and the concept of Service Function Chaining. Then we focused on a description and performance evaluation of our Flexible Compute Virtualization based on Unikernels and library OSs, which focuses on lightweight and rich execution environments.

All of those components are working together with the various VirIoT core services (ThingVisors, vSilos, vThings, etc.), leading to more advanced implementations (e.g. ThingVisors and vSilo Flavours using FogFlow, Unikernels, etc.) that achieve distributed deployments on edge nodes together with lightweight compute virtualization.

References

[1] Kubernetes Production-Grade Container Orchestration. [Online]. Available: https://kubernetes.io/

[2] VirIoT open-source code. [Online]. Available: https://github.com/fed4iot/VirIoT

[3] NEC Labs Europe, "FogFlow tutorial," https://fogflow.readthedocs.io/, (Accessed November 26th 2019).

[4] H. Topcuoglu, S. Hariri, and Min-You Wu, "Performance-effective and low-complexity task scheduling for heterogeneous computing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 13, no. 3, pp. 260–274, March 2002.

[5] H. Arabnejad and J. G. Barbosa, "List scheduling algorithm for heterogeneous systems by an optimistic cost table," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 3, pp. 682–694, March 2014.

[6] M. C. Luizelli, W. L. da Costa Cordeiro, L. S. Buriol, and L. P. Gaspary, "A fix-and-optimize approach for efficient and large scale virtual network function placement and chaining," *Computer Communications*, vol. 102, pp. 67 – 77, 2017.

[7] D. Bhamare, M. Samaka, A. Erbad, R. Jain, L. Gupta, and H. A. Chan, "Multi-objective scheduling of micro-services for optimal service function chains," in *2017 IEEE International Conference on Communications (ICC)*, May 2017, pp. 1–6.

[8] L. Wang, Z. Lu, X. Wen, R. Knopp, and R. Gupta, "Joint optimization of service function chaining and resource allocation in network function virtualization," *IEEE Access*, vol. 4, pp. 8084–8094, 2016.

[9] M. T. Beck and J. F. Botero, "Scalable and coordinated allocation of service function chains," *Computer Communications*, vol. 102, pp. 78 – 88, 2017.

[10] J. Martins, M. Ahmed, C. Raiciu, V. Olteanu, M. Honda, R. Bifulco, and F. Huici, "ClickOS and the art of network function virtualization," in *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'14. USENIX, 2014, pp. 459–473. [Online]. Available: http://dl.acm.org/citation.cfm?id=2616448.2616491

[11] I. Marinos, R. N. Watson, and M. Handley, "Network Stack Specialization for Performance," in *Proceedings of the 2014 ACM Conference on Computer Communication*, ser. SIGCOMM '14. New York, NY, USA: ACM, 2014, pp. 175–186. [Online]. Available: http://doi.acm.org/10.1145/2619239.2626311

[12] S. Kuenzer, A. Ivanov, F. Manco, J. Mendes, Y. Volchkov, F. Schmidt, K. Yasukata, M. Honda, and F. Huici, "Unikernels everywhere: The case for elastic cdns," in *Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, ser. VEE '17. New York, NY, USA: ACM, 2017, pp. 15–29. [Online]. Available: http://doi.acm.org/10.1145/3050748.3050757

[13] Galois Inc., "The haskell lightweight virtual machine (halvm)," https://github.com/GaloisInc/HaLVM, 2008, online; accessed Jan, 25 2021.

[14] A. Madhavapeddy and D. J. Scott, "Unikernels: Rise of the Virtual Library Operating System," *Queue*, vol. 11, no. 11, pp. 30:30–30:44, Dec. 2013. [Online]. Available: http://doi.acm.org/10.1145/2557963.2566628

[15] runtimejs.org, "JavaScript Library Operating System for the Cloud," http://runtimejs.org/, online; accessed Jan, 25 2021.

[16] GitHub, "Erlang on Xen," https://github.com/cloudozer/ling, online; accessed Jan, 25 2021.

[17] A. Randazzo and I. Tinnirello, "Kata containers: An emerging architecture for enabling mec services in fast and secure way," in *Proceedings of the 6th International Conference on Internet of Things: Systems, Management and Security*, ser. IOTSMS'19. IEEE, 2019, pp. 209–214.

[18] A. Van de Ven, "An introduction to Clear Containers," https://lwn.net/Articles/644675/, online; accessed Jan, 25 2021.

[19] Habana, "100% AI," https://habana.ai/, online; accessed Jan, 25 2021.

[20] Google, "Cloud TPU," https://cloud.google.com/tpu, online; accessed Jan, 25 2021.

[21] Intel, "Intel® Movidius™ Vision Processing Units (VPUs)," https://www.intel.com/content/www/us/en/products/processors/movidius-vpu.html, online; accessed Jan, 25 2021.

[22] F. Manco, C. Lupu, F. Schmidt, J. Mendes, S. Kuenzer, S. Sati, K. Yasukata, C. Raiciu, and F. Huici, "My vm is lighter (and safer) than your container," in *Proceedings of the 26th Symposium on Operating Systems Principles*, ser. SOSP '17. New York, NY, USA: ACM, 2017, pp. 218–233. [Online]. Available: http://doi.acm.org/10.1145/3132747.3132763

[23] A. Madhavapeddy, T. Leonard, M. Skjegstad, T. Gazagnaire, D. Sheets, D. Scott, R. Mortier, A. Chaudhry, B. Singh, J. Ludlam, J. Crowcroft, and I. Leslie, "Jitsu: Just-In-Time Summoning of Unikernels," in *12th USENIX Symposium on Networked Systems Design and Implementation*, ser. NSDI '15. Oakland, CA: USENIX Association, 2015, pp. 559–573. [Online]. Available: https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/madhavapeddy

[24] A. Kantee, "Flexible operating system internals: The design and implementation of the anykernel and rump kernels," Ph.D. dissertation, Aalto University, 2012.

[25] H.-C. Kuo, D. Williams, R. Koller, and S. Mohan, "A linux in unikernel clothing," in *Proceedings of the Fifteenth European Conference on Computer Systems*, ser. EuroSys '20. New York, NY, USA: Association for Computing Machinery, 2020. [Online]. Available: https://doi.org/10.1145/3342195.3387526

[26] Google, "Protocol buffers - google's data interchange format." [Online]. Available: https://github.com/protocolbuffers/protobuf

[27] Docker, "Docker: Enterprise Container Platform ," https://www.docker.com.

[28] Cloud Foundary, "Open Source Cloud Application Platform ," https://www.cloudfoundry.org.

[29] OCI Image Format Specification, https://github.com/opencontainers/image-spec.

[30] OCI Initiative, "OCI Image Format Specification ," https://www.opencontainers.org.

[31] J. Thalheim, P. Bhatotia, P. Fonseca, and B. Kasikci, "Cntr: Lightweight OS containers," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. Boston, MA: USENIX Association, 2018, pp. 199–212. [Online]. Available: https://www.usenix.org/conference/atc18/presentation/thalheim

[32] E. Oakes, L. Yang, D. Zhou, K. Houck, T. Harter, A. Arpaci-Dusseau, and R. Arpaci-Dusseau, "SOCK: Rapid Task Provisioning with Serverless-Optimized Containers," in *Proceedings of the USENIX Annual Technical Conference (USENIX ATC'18)*, 2018.

[33] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. Swift, "Peeking behind the curtains of serverless platforms," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, 2018, pp. 133–146.

[34] R. Koller and D. Williams, "Will Serverless End the Dominance of Linux in the Cloud?" in *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, ser. HotOS '17. New York, NY, USA: ACM, 2017, pp. 169–173. [Online]. Available: http://doi.acm.org/10.1145/3102980.3103008

[35] A. Madhavapeddy, R. Mortier, C. Rotsos, D. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, and J. Crowcroft, "Unikernels: Library operating systems for the cloud," *Acm Sigplan Notices*, vol. 48, no. 4, pp. 461–472, 2013.

[36] A. Kivity, D. Laor, G. Costa, P. Enberg, N. Har'El, D. Marti, and V. Zolotarov, "Osv—optimizing the operating system for virtual machines," in *2014 {USENIX} Annual Technical Conference ({USENIX}{ATC} 14)*, 2014, pp. 61–72.

[37] D. Williams, R. Koller, M. Lucina, and N. Prakash, "Unikernels as processes," in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SoCC '18. New York, NY, USA: ACM, 2018, pp. 199–211. [Online]. Available: http://doi.acm.org/10.1145/3267809.3267845

[38] P. Olivier, D. Chiba, S. Lankes, C. Min, and B. Ravindran, "A binary-compatible unikernel," in *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, ser. VEE 2019. New York, NY, USA: ACM, 2019, pp. 59–73. [Online]. Available: http://doi.acm.org/10.1145/3313808.3313817

[39] J. Dike, "User Mode Linux," in *Proceedings of the 5th Anual Linux Showcase and Conference*, ser. ALS'01. USENIX Association, 2001, pp. 3–14.

[40] The OpenStack Foundation, "Kata Containers," https://katacontainers.io/, (Accessed Aug 15th 2018).

[41] Google Inc., "gVisor: Container Runtime Sandbox," https://github.com/google/gvisor, (Accessed May 8th 2018).

[42] IBM, "Nabla Containers," https://github.com/nabla-containers/runnc, (Accessed July 3rd 2019).

[43] Ixia, "IxANVL," https://ixia.keysight.com/resources/ixanvl-overview, (Accessed Sep 14th 2018).